Universität Stuttgart
Fakultät Informatik

Prüfer:	Prof. Dr. Hans-Joachim Wunderlich
Betreuer:	Dipl.-Phys. Rainer Dorsch
Beginn am:	18.09.2000
Beendet am:	23.02.2001
CR-Klassifikation:	C.3 C.5.4

Diplomarbeit-Nr. 1871

Digitales Diktiergerät als System-on-a-Chip mit FPGA-Evaluierungsboard

Daniel Bretz

Institut für Informatik
Breitwiesenstr. 20-22
D-70565 Stuttgart

Kurzfassung

Ziel der Arbeit war es, eine System-on-a-Chip Entwicklungsplattform für die Abteilung Rechnerarchitektur für Forschung und Lehre bereitzustellen und exemplarisch ein digitales Diktiergerät als Demonstrator mit der Plattform zu implementieren.

Für die Arbeit wurde als Grundsystem die von der European Space Agency (ESA) entwickelte LEON-Plattform für eingebettete Systeme ausgewählt. Kern des Open Source Systems ist ein SPARC V8 kompatibler Prozessor, der über einen AMBA-Bus mit den übrigen Komponenten kommuniziert. LEON wurde an die Entwicklungswerkzeuge der Abteilung und an ein ausgewähltes Prototypen-Evaluierungsboard angepasst. Auf der fertigen Plattform wurde das digitale Diktiergerät implementiert und erfolgreich auf der Hardware in Betrieb genommen.

Danksagung

Die vorliegende Diplomarbeit ist im Zeitraum Herbst, Winter 2000/2001 entstanden. Ich möchte mich an dieser Stelle bei allen bedanken, die mir dabei geholfen haben.

Namentlich erwähnen möchte ich Student Jens Künzer und Dipl.-Phys. Stefan Gerstendörfer, die mir bei vielen technischen Fragen helfend zur Seite standen. Besonderen Dank geht an meinen Betreuer Dipl.-Phys. Rainer Dorsch für das Korrekturlesen und der zu jeder Zeit moralische Unterstützung und gute Ratschläge gab. Bei Prof. Dr. Wunderlich bedanke ich mich für die Ermöglichung dieser Arbeit, zu welcher auch die Anschaffung der Hardware zählt. Dank auch an Jiri Gaisler, Hauptentwickler von LEON bei der ESA, und Dr. Vanden Bout bei Xess für den guten Service bei allen Fragen, die das Board betrafen. Zuletzt Dank an meine Eltern, die mir über die stressige Zeit der Diplomarbeit geholfen und das Studium ermöglicht haben.

Inhaltsverzeichnis

1 Einleitung **7**
- 1.1 System-on-a-Chip Entwicklung . . . 7
- 1.2 Evaluierungshardware . . . 9
- 1.3 Die Arbeit . . . 9

2 Die LEON-Plattform **11**
- 2.1 LEON ein Open Source Projekt . . . 11
  - 2.1.1 Aufbau der LEON-Plattform . . . 12
  - 2.1.2 Struktur des Gesamtsystems . . . 13
  - 2.1.3 Der Prozessorkern . . . 14
- 2.2 Paketumfang von LEON . . . 14
- 2.3 Struktur der Quelltexte und Verzeichnisse . . . 15
- 2.4 Simulation auf Registertransferebene . . . 18
  - 2.4.1 Inhalt der Testbench . . . 19
  - 2.4.2 Simulation von Standardsoftware . . . 20
- 2.5 Synthese mit SynopsysDC . . . 20
- 2.6 Xilinx-Designflow . . . 21
- 2.7 Simulation auf Gatterebene . . . 22
- 2.8 Konfiguration der Plattform . . . 25
- 2.9 Programmierung und Ausführung . . . 26
- 2.10 AMBA . . . 27
  - 2.10.1 APB . . . 29
  - 2.10.2 AHB . . . 29

3 Prototyping von LEON mit XSV-800 Board **33**
- 3.1 Funktionalität des Boards . . . 33
- 3.2 PLDs . . . 35
- 3.3 FPGA . . . 35
- 3.4 CPLD . . . 36
- 3.5 Kommunikation und Konfiguration . . . 37
- 3.6 CPLD Synthese . . . 39

3.7 Synthese von LEON für XSV-800 40
3.8 Betrieb des XSV-800 Board 42

4 Diktiergerät als SoC **45**
4.1 Entwicklungszyklus 45
4.2 Funktionsbeschreibung 48
4.3 Schaltungsaufbau 49
4.3.1 Die Signale des AKM AK4520A 49
4.3.2 APB Slave 50
4.3.3 Mikrotaster und Hexdisplay 51
4.3.4 Timer 51
4.3.5 Audio Ansteuerung 52
4.3.6 AHB Master 54
4.4 Integration in die LEON-Plattform 55
4.5 Aufbau und Benutzung der Software 55
4.5.1 Zugriff auf die Register und ihre Funktion 56
4.5.2 Die Funktionen der Software 56
4.5.3 Benutzung der Software 59

5 Zusammenfassung und Ausblick **61**

A CVS **63**
A.1 Umgang mit CVS 63
A.2 Die Projekte der Entwicklungsumgebung 65

B ModelSIM **67**

C Portierung der XESS Software nach LINUX **68**
C.1 Verzeichnisstruktur 68
C.2 Änderungen des Quelltextes im Detail 69

D Datei Auszüge **71**
D.1 target.vhd 71
D.2 LEON Top-Entity 76
D.3 ddm.h 77

Abbildungsverzeichnis

2.1 Übersicht über das Gesamtsystem von LEON 13
2.2 Übersicht über die Schnittstellen- und Konfigurationsdateien . . . 16
2.3 Übersicht der LEON-Quelldateien 17
2.4 Designflow der Plattform 24
2.5 Vereinfachte Darstellung des AMBA-Bus 28
2.6 Simpler Datentransfer auf AHB 31
2.7 Simpler Datentransfer mit optimalem Zeitverhalten 32

3.1 XSV-800 Boardübersicht 38

4.1 Entwicklungszyklus einer Anwendung auf der Plattform 47
4.2 Übersicht über den DDM-Core 48
4.3 Das Zeitverhalten der Taktsignale des AK4520A 50
4.4 Schaltung des APB Slave 50
4.5 Schaltung Mikrotaster und Hexdisplay 51
4.6 Schaltung des Timers 52
4.7 Schaltung der Audio Ansteuerung 53
4.8 Schaltung des AHB-Master 54
4.9 Belegung und Adressen der Speicher abgebildeten Register 56
4.10 Bedienung des Diktiergeräts 60

Kapitel 1

Einleitung

Menschen haben heute im Alltag an vielen Stellen mit elektronischen Schaltkreisen zu tun, ob bewusst oder unbewusst. Meist sind dies eingebettete Systeme, also Schaltungen, die in einem Gesamtsystem eingebettet sind und die Aufgaben der Steuerung oder Signalverarbeitung übernehmen. Man findet sie zum Beispiel bei Kraftfahrzeugen im Antiblockiersystem (ABS), der Motorsteuerung, dem Radio, der Klimaanlage und vielem mehr. Gerade in der Signalverarbeitung werden dabei spezielle Funktionen in die Schaltung integriert. Die Umsetzung der Systemlösungen erfolgt in eingebetteten Systemen oft als Mischung von Hardware und Software.

1.1 System-on-a-Chip Entwicklung

Die heutigen Integrationsdichten ermöglichen es, viele verschiedene Funktionen auf einem Chip unterzubringen. Damit kann ein System, das früher in mehreren Chips auf einer Platine untergebracht war, in einem einzigen Chip integriert werden. Dies erhöht die Geschwindigkeit der Kommunikation der Schaltungen untereinander und senkt den Energieverbrauch. Das System braucht weniger Platz und die Herstellung eines einzigen Chips ist preisgünstiger. Ein solches auf einem Chip integriertes System ist ein System-on-a-Chip (SoC) [MK98]. Um den Softwareteil des Systems auszuführen, wird ein Prozessor benötigt, der meist auch die Kontrolle des Gesamtsystems übernimmt. Prozessoren, die in SoC oder eingebetteten Systemen zum Einsatz kommen, unterscheiden sich gegenüber den in Computern üblich verwendeten CPUs in einigen Dingen. So spielt bei ihnen die Leistung nur eine untergeordnete Rolle, da die eigentlichen Aufgaben von den anderen Cores ausgeführt werden. Sie haben einen geringen Energieverbrauch, was in mobilen Geräten unerlässlich ist. Zur Kommunikation mit den anderen Cores und der Außenwelt benötigen sie einen on-Chip Bus und beinhalten meist eine

größere Anzahl an Standardschnittstellen, wie UART und parallele Schnittstelle. Viele bieten zur besseren Programmierung die Möglichkeit der Instruktionserweiterung um selbst definierte Coprozessorbefehle und die Unterstützung von komprimierten Befehlssatz.

Die Entwicklung eines kompletten Systems auf einem Chip wurde erst durch die immer besseren Fertigungsverfahren in der Chiptechnik und der dadurch immer größeren Anzahl von Transistoren pro Chip möglich. Die immer größere Funktionalität der Schaltungen bringt aber auch Probleme mit sich. Die Anzahl der Fehlermöglichkeiten auf einem Chip steigt mit der Zahl der auf ihm aufgebrachten Bauelemente. Der Entwurf immer größerer Schaltungen benötigt mehr Zeit, wobei aber gerade die Produktlebenszyklen immer kürzer werden.

Um die Time-to-Market zu verkürzen oder zumindest konstant zu halten, werden bei SoCs neue Entwicklungsverfahren angewandt. Die einzelnen Komponenten des Systems werden getrennt entwickelt und bleiben in mehrere funktionale Einheiten, Cores oder Macros genannt, unterteilt. Diese werden wiederverwendbar ausgelegt. Die Entwickler können so bei einem neuen System auf alte Cores zurückgreifen. Dazu wird eine schon vorhandene Grundplattform verwendet, an die ein speziell für die Aufgabe entwickelter Core abgebunden wird. Um die Normen und Entwicklungsvorschriften der Cores kümmert sich die Virtual Socket Interface Alliance[VSI97], die VSIA. Bei den Entwicklern muss zwischen den Core Designern, welche die Schaltung für den Core entwerfen und den System-Integratoren, die ein System aus einzelnen Cores zusammenstellen, unterschieden werden. Bei den Cores unterscheidet man zwischen Hard-, Soft- und Firmcores.

Hardcores sind Macros, die als komplett verdrahtetes Layout vorliegen und an eine Herstellungstechnik gebunden sind. Für sie gibt es die genauen physikalischen Eigenschaften und sie sind vollständig verifiziert. Der Integrator erhält bei diesem Typ eine Spezifikation, ein Simulationsmodell und die Schnittstellenbeschreibung. Auf höheren Abstraktionsebenen ist die Implementierung des Cores nur dem Hersteller bekannt. Die Anbieter von Hardcores sind deswegen meist Chipproduzenten, bei denen das komplette SoC gefertigt wird.

Bei einem Softcore bekommt der SoC Integrator den kompletten Quelltext der Schaltung und die dazugehörigen Testbenches. Dadurch kann er die Schaltung besser in sein Design integrieren und eventuell daran anpassen. Im Gegensatz zum Hardcore muss die Schaltung den kompletten Designflow durchlaufen, was im Zusammenspiel mit den verwendeten Synthesewerkzeugen und dem Zeitverhalten Probleme bereiten kann. Ein Softcore sollte deswegen möglichst ein breites Spektrum an Synthesewerkzeugen und möglichst die beiden Hardwarebeschreibungssprachen, VHDL und Verilog, unterstützen. Ein Problem der Softcores ist der Schutz des Intellectual Property (IP), des geistigen Eigentums.

Firmcores sind zwischen Soft- und Hardcores einzuordnen. Der Integrator erhält nicht die kompletten Quelltexte, kann aber bestimmte Designparameter vor-

geben. Dies können z.B. verschiedene Bus- oder Registerbreite, optionale Funktionalität oder unterschiedliche Speicherausstattung sein. Firmcores werden meistens in Registertransferlevel (RTL) oder als Netzliste auf Gatterebene ausgeliefert, dabei eine gute Vorhersage über die physikalischen Eigenschaften gemacht werden [MK98].

1.2 Evaluierungshardware

Ein Application Specific Integrated Circuit (ASIC), also eine anwendungsspezifische Schaltung, kostet in der Entwicklung sehr viel Geld. Zwar lässt sich damit die größte Leistung erreichen, doch ist sie für einige Schaltungsentwürfe wegen des großen Entwicklungsaufwandes zu unrentabel. Um die Entwicklungskosten der ASICs zu senken, werden neue Wege in der Entwicklung gegangen. Dabei werden vor allem die hohen Kosten der Maskenerstellung für die Prototypen umgangen, indem man die Schaltung auf andere Art validiert.

Um Schaltungen auf einem Chip unterzubringen, ohne einen ASIC-Entwurf zu benutzen, gibt es die programmierbaren Bausteine (programmable logic device, PLD). Mit der Zeit hat sich ein ganzes Spektrum von diesen Bausteinen entwickelt. Die höchste Entwicklungsstufe sind die FPGAs. Auf ihnen können sehr komplexe Schaltungen, wie z.B ein Mikroprozessor, abgebildet werden. Größte Verwendung finden die FPGAs in der Hardwarevalidierung, die auf zwei Arten erfolgen kann. Zum einen werden die FPGAs in Simulationsumgebungen zur schnellen Berechnung von logischen Funktionen verwendet. Dazu werden mehrere FPGAs in einem System parallel betrieben. Die andere Möglichkeit ist, eine Schaltung komplett als Prototyp auf ein einziges FPGA abzubilden, was durch die heute sehr komplexen Bausteine möglich ist. Die abgebildete Schaltung kann im Gegensatz zur reinen Simulation mit hohen Geschwindigkeiten betrieben werden. Wenn die Funktionalität der Schaltung validiert ist, wird sie auf die eigentlichen Zieltechnologie abgebildet. Die Entwicklungszeit wird dadurch reduziert und die Entwicklung kostet weniger Geld (rapid Prototyping). Auf dem Markt befinden sich heute eine große Anzahl verschiedener Produkte zur Hardwareverifikation mit FPAGs.

1.3 Die Arbeit

In der vorliegenden Arbeit wurde eine Plattform zur System-on-a-Chip Entwicklung für die Abteilung Rechnerarchitektur zusammengestellt. Diese soll der Abteilung als Lehrobjekt und als Forschungsumgebung, an welcher theoretische Forschungsergebnisse an praxisnahen Schaltungen verifiziert werden können, die-

nen. Für die Arbeit wurden verwendbare Komponenten gesucht und ausgewählt. Als Beispielanwendung wurde ein digitales Diktiergerät implementiert. Um eine schnelle Entwicklung zu ermöglichen und die Schaltung in Echtzeit betreiben zu können, wurde ein FPGA basiertes Hardware-Evaluierungsboard ausgewählt und in die Entwicklungsumgebung integriert.

Als Grundsystem dient die in dem European Space research and TEchnology Centre (ESTEC) der European Space Agency (ESA) entwickelte LEON-Plattform. Die Plattform enthält einen eingebetteten Prozessor mit Programmierumgebung. Es entstand folgende Entwicklungsumgebung, die sich grob in vier Bereiche einteilen lässt: der Quelltext der LEON-Plattform mit Syntheseskripten, die Softwareentwicklungsumgebung, die Test- und Simulationsumgebung und das Evaluierungsboard. Eine klare Trennung ist allerdings nicht möglich, da alle Bereiche ineinander übergreifen.

In Kapitel 2 wird die Grundplattform beschrieben. Zu dem Quelltext des Prozessors gehören neben den eigentlichen Codedateien auch die Syntheseskripte und die Konfigurationsdateien. Eng damit verbunden sind Dateien für die Simulation und die Testbench. Als Synthesewerkzeug wird der Sysnopsys Design Compiler verwendet. Als Softwareentwicklungsumgebung wird der von der ESA/ESTEC gelieferte C/C++/ADA Compiler[1] benutzt. Dazu gehört auch ein SPARC (LEON) Instruktionslevelemulator und einige zusätzliche Werkzeuge für die Softwareentwicklung des LEON. Die Test- und Simulationsumgebung enthält hauptsächlich die Testbench für die CPU, aber auch die Möglichkeit, die CPU mit selbst übersetzten Programmen zu simulieren. Zur Simulation wird Mentor ModelSIM verwendet.

Als Hardwareevaluierungsboard dient das XSV-800 Board der Firma Xess. Die Integration der Plattform darauf und die Benutzung wird in Kapitel 3 beschrieben. Auf ihm befinden sich ein Xilinx Virtex XCV800 FPGA, ein Xilinx XC95108 CPLD, 2 MB SRAM, 1 MB FlashRAM und viele Peripherieschnittstellen. Zu dem Board gehört auch die für seine Programmierung zuständige Software.

In Kapitel 4 wird die Umsetzung des Diktiergeräts auf die Plattform beschrieben. Dazu gehört die Entwicklung eines eigenen Cores, welcher über den AMBA-Bus mit dem System verbunden ist. Das System ist zu seiner Anbindung entsprechend angepasst. Spezielle Software integriert die Funktionen des Diktiergeräts darauf. Am Ende des Kapitels wird der Umgang mit dem Diktiergerät erklärt.

[1] beruhend auf dem GNU Compiler und den GNU Binutils

Kapitel 2

Die LEON-Plattform

Die Steuerung von eingebetteten Systemen kann durch eine CPU erfolgen. Dabei ist der Zugriff auf den Quelltext des Prozessors für die Abteilung Rechnerarchitektur von großem Interesse und sollte deswegen als Softcore vorliegen. Dank des geringen Kostenaufwandes einer FPGA-Entwicklung und den Vorbildern aus der Softwareentwicklung gibt es mittlerweile drei ernst zu nehmende Open Source Projekte für CPUs. Diese sind die Freedom-, die OpenRISC- und die LEON-CPU. Zur Zeit dieser Arbeit befinden sich die Freedom- und die OpenRISC-CPU noch am Anfang ihrer Entwicklung, weswegen hier der von der ESA entwickelte LEON-Prozessor verwendet wird. Der Vorgänger von LEON ist die ERC32-CPU, von welcher ebenfalls die Quelltexte bei der ESA frei erhältlich sind. Diese ist aber eine eigenständige CPU, die sich nicht sehr gut für eingebettete Systeme eignet. Kommerzielle Alternativen wurden nicht berücksichtigt, da der Kostenaufwand für ein solches System mit Zugriff auf den Quelltext zu groß wäre. Die Kosten würden neben der Beschaffung des Systems auch einen eigenen Rechnercluster beinhalten. Es würde zudem zu lizenzrechtlichen Problemen bei Lehrveranstaltungen und bei der Veröffentlichung von Forschungsergebnissen kommen. Kommerzielle Alternativen sind der ARM, der Tensilica- oder ARC-Prozessor. Die beiden letzteren sind Entwicklungsumgebungen, bei denen die CPUs aus Softcores bestehen. Der Entwickler kann die CPU in vielen Parametern konfigurieren. Nach Auswahl erzeugen die Entwicklungswerkzeuge eine Beschreibung der Schaltung mit dem dazugehörigen Compiler.

2.1 LEON ein Open Source Projekt

Die verwendete LEON SoC-Plattform [Gai00a] ist ein Open Source Projekt der ESA/ESTEC[1]. Das bedeutet, dass der gesamte VHDL-Quelltext des Systems er-

[1] european space agency/european space research and technology centre

hältlich ist und frei benutzt und verändert werden darf. Die ESA/ESTEC entwickelt Prozessoren für den Einsatz in Satelliten. Für den Einsatz in dieser Prozessor unfreundlichen Umgebung gibt es eine Fehler tolerante Version von LEON. Diese wird von der ESA kommerziell vertrieben. In der freien Version von LEON ist der Fehler tolerante Teil entfernt. Die Open Source Entwicklung bringt für die ESA drei wesentliche Vorteile. Durch die höhere Benutzung werden die Funktionen der CPU besser validiert. Der Funktionsumfang und die Leistung werden ohne eigene Entwicklungsbemühungen vergrößert. Für die Entwicklung der Plattform entstehen neue, meist freie, Werkzeuge und lauffähige Software. Ein Beispiel hierfür ist μCLinux, welches schon auf die Plattform portiert wird.

LEON, als auch der Vorgänger ERC32, implementieren den SPARC Befehlssatz. Jedoch unterstützt LEON die etwas neuere Version 8, im Gegensatz zum ERC32, welcher die Version 7 implementiert. LEON ist eine komplette Neuentwicklung der ESA und wurde als eingebettete Plattform ausgelegt, worin sie sich hauptsächlich von dem ERC32 unterscheidet. Bemerkbar macht sich dies an den integrierten Schnittstellen - wie UARTs, parallele I/O Schnittstelle, PCI - und vor allem an der Schnittstelle für Coprozessoren und dem Systembus, an dem UDF-Cores angeschlossen werden können. Der Quelltext der Hardware des LEON stehen unter der LGPL, d.h., dass alle Änderungen, die die ursprüngliche Plattform betreffen, veröffentlicht werden müssen. Eigene entwickelte Cores, die an dem System angeschlossen werden, müssen aber nicht veröffentlicht werden. Der restliche Quelltext von LEON, wie z.B. der der Testbench stehen unter der GPL[2] oder sind wie die des Instruktionslevelemulators nicht erhältlich.

2.1.1 Aufbau der LEON-Plattform

Im Februar 2000 wurde LEON in einer ersten Version der Öffentlichkeit vorgestellt und zum Herunterladen von dem ESA-Server [leo01] angeboten. Es war die LEON-1 Version 2.0. Auf diese folgte eine vor allem Fehler korrigierte und leicht verbesserte Version 2.1. Die Diplomarbeit beruht auf der LEON-1 Version 2.2, welche seit Okt/Nov 2000 erhältlich ist. Seither wird die von ARM spezifizierte Advanced Microcontroller Bus Architecture (AMBA) benutzt. LEON ist mit der Scalable Processor ARChitecture (SPARC) in der Version 8 [SPA92] kompatibel. Diese als IEEE-P1754 definierte Spezifikation ist eine frei benutzbare 32 Bit Architektur, welche in Bezug auf Registeranzahl und CPU Umfang skalierbar ist. Das bedeutet, dass Einheiten wie MMU, FPU, Co-Prozessoren, Caches optional integriert werden können, aber deren Ansteuerung im Falle einer Implementierung klar definiert ist.

[2] general public license

2.1.2 Struktur des Gesamtsystems

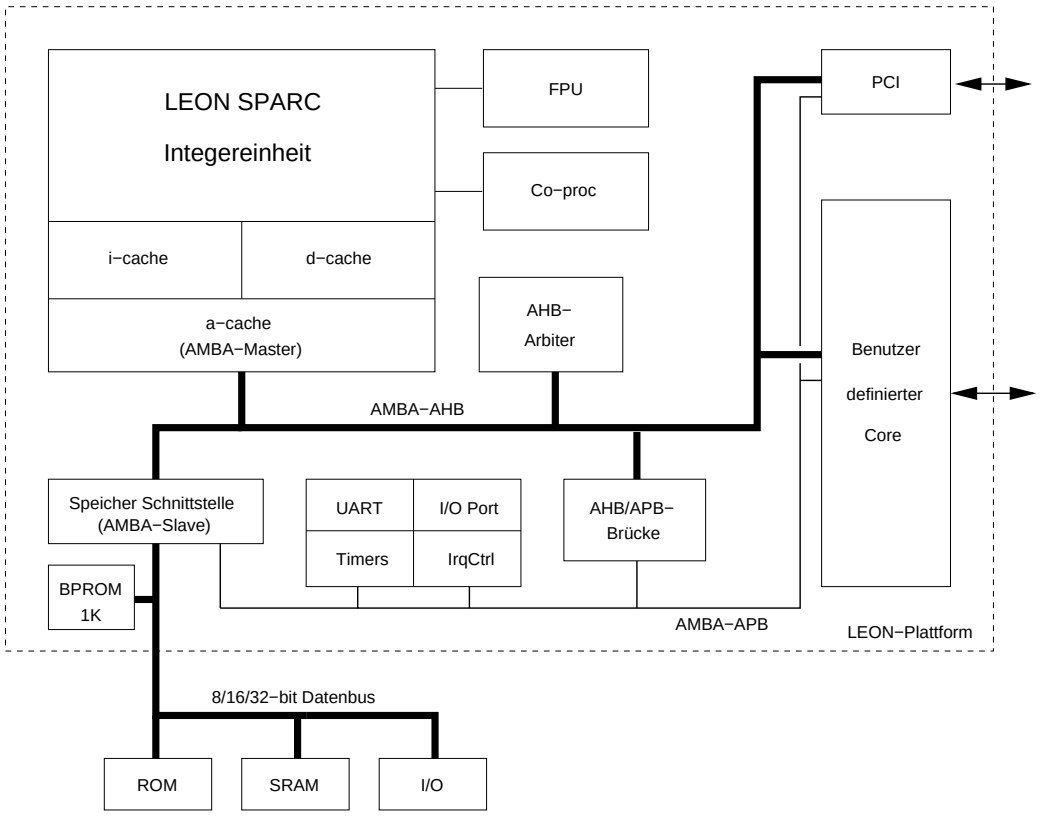

Abbildung 2.1: Übersicht über das Gesamtsystem von LEON

Abb. 2.1 zeigt eine Übersicht über den Aufbau der LEON-Plattform. Oben links befindet sich der eingebettete Prozessor, welcher im nächsten Abschnitt näher beschrieben wird. Der Prozessor ist über den Befehls- und den Datencache mit dem AMBA-Bus verbunden und über diesen auch mit den anderen Modulen oder Cores des Systems. Die Plattform besitzt von Haus aus eigene Peripherie. Dies sind zwei UARTs, eine parallele Schnittstelle, ein Interruptcontroller, zwei Timer und ein Watchdog. Sie sind in der Mitte der Zeichnung ersichtlich. Gesteuert werden sie über Speicher abgebildete (memory mapped) Register, welche über die AHB/APB-Brücke des AMBA-Bus mit dem Prozessor verbunden sind. Der AMBA-Bus und seine Komponenten werden in Abschnitt 2.10 näher beschrieben. Links neben der Peripherie ist die Speicherschnittstelle zu sehen. Diese erhält als AMBA-Slave von den AMBA-Mastern, z.B. den Caches, Datentransferaufrufe zum Speicher. Die Datenbreite des Speicherbus kann dabei wahlweise 8, 16 oder

32 Bit sein. Dies wird über ein Konfigurationsregister, jeweils für RAM, ROM und I/O getrennt, gesteuert und kann auch während des Betriebs umgeschaltet werden. Ab der Version 2.2 des LEON kann auch ein internes BPROM (Boot PROM) mit in das Design synthetisiert werden. In das 1 KByte große ROM passt ein Bootprogramm, welches die Speicherkonfiguration automatisch setzt und anschließend über die serielle Schnittstelle ein auszuführendes Programm in den Speicher laden kann. Der AMBA-Arbiter ist für die Kontrolle des AMBA-Bus zuständig. PCI ist kein komplettes Gerät, sondern dient als Schnittstelle zu einer PCI-Schnittstelle. An ihr kann ein PCI-Core von Phoenix angeschlossen werden. Unten rechts in der Abbildung ist ein benutzerdefinierter Core zu sehen. Er zeigt, wie fremde Schaltungen in die LEON-Plattform integriert werden können.

2.1.3 Der Prozessorkern

Die Integereinheit zusammen mit den Caches bilden den Kern des LEON-Cores. Sie arbeitet nach SPARC-Definition mit einer Breite von 32 Bit. Die Pipeline ist 5-stufig ausgelegt und besteht aus der Standardabfolge instruction fetch, instruction decode, execute, memory, write back mit 1 delay slot, gemäß SPARC Norm. Zum Multiplizieren steht ein iterativer Multiplizierer zur Verfügung, der mit in das Design synthetisiert werden kann. Wegen des hohen Platzverbrauchs ist er in FPGA-Designs normalerweise nicht integriert. Ein Dividierer gibt es in der bisherigen Version noch nicht. Für Fließkommaanwendungen gibt es eine FPU Schnittstelle zum FPU Meiko-CORE, welcher kommerziell bezogen werden kann. Der Prozessor bietet die Möglichkeit, Coprozessoren anzuschließen. Damit können hinzugefügte Cores mit eigenen Befehlen gesteuert werden.

Die CPU besitzt einen Daten- und einen Befehlscache. Beide sind direkt abbildend ausgelegt und ihre Größe ist von 1-64 kByte konfigurierbar. Das Registerfile enthält alle notwendigen und die von SPARC typischen Registerwindows. Registerwindows ermöglichen bei Funktionsaufrufen überlappende Bereiche von Registern, in denen die Parameter der Funktionen stehen. Sie sind näher in Kapitel 4 von [SPA92] beschrieben. Die Anzahl der Windows ist von 2-32 konfigurierbar und bei den Designs für FPGAs auf 8 gesetzt.

2.2 Paketumfang von LEON

Bei der ESA sind für das LEON-Projekt zwei Pakete für die Entwicklung erhältlich. Erstens der Quelltext für die LEON-Plattform in VHDL und zweitens eine Softwareentwicklungsumgebung, die wahlweise für Solaris oder Linux angeboten wird. Im Detail enthalten die Pakete folgende Bestandteile:

Quelltext:

- VHDL Code für die LEON-Plattform
- Makefiles und Buildskripte für die Synthese und Simulation
- Eine Testbench mit Quelltext und die dazugehörige Simulationsumgebung für Speicher und Datenauswertung
- bprom Generator; Quelltext für ein internes BootROM und ein Programm, dass eine VHDL-Beschreibung aus dem ROM erzeugt

Softwareentwicklung:

- GNU C/C++/Ada-Crosscompiler [Gai99],[Sta98]
- GNU Binutils [RHP93]
- Cygnus/Newlib standalone C-Bibliothek
- RTEMS Kernel [On-98] mit LEON Unterstützung
- mkprom Werkzeug; erzeugt aus den vom Compiler erzeugten Binaries ein ROM
- SIS LEON Instruktionslevelemulator
- DDD als Graphische Benutzungsoberfläche für gdb

2.3 Struktur der Quelltexte und Verzeichnisse

In diesem Abschnitt wird der Aufbau der Verzeichnisse und Dateien, des LEON-Projekts beschrieben. Zu dem, von der ESA bezogenen Quelltext, sind noch weitere bei der Arbeit entstandene Verzeichnisse dazugekommen. Nach dem Entpacken des LEON Quelltext-Archivs von der ESA oder dem Aushecken der Sourcen aus dem CVS-Projekt *Leon*-1; siehe Anhang *A.2;* bekommt man ein Verzeichnis mit folgenden Unterverzeichnissen:

leon:	die VHDL-Quelltexte von LEON
doc:	in diesem ist das LEON-Handbuch [Gai00a] zu finden
tbench:	die Testbenchumgebung für ModelSIM in VHDL
syn:	das Syntheseverzeichnis mit Skripten für die verschiedenen Synthesewerkzeuge

tsource: die Quellen für die Software der Testbench

bprom: der Quelltext des BPROM und ein ROM Generator, welcher eine VHDL Beschreibung aus dem ROM erzeugt

Im CVS-Projekt sind noch folgende bei dieser Arbeit dazugekommenen Verzeichnisse zu finden:

tenv: eine von der Testbench abgeleitete Simulationsumgebung um Standardprogramme auszuführen

tenv32_back: spezielle Simulationsumgebung für die Simulation der XSV-800 Designs auf Gatterebene

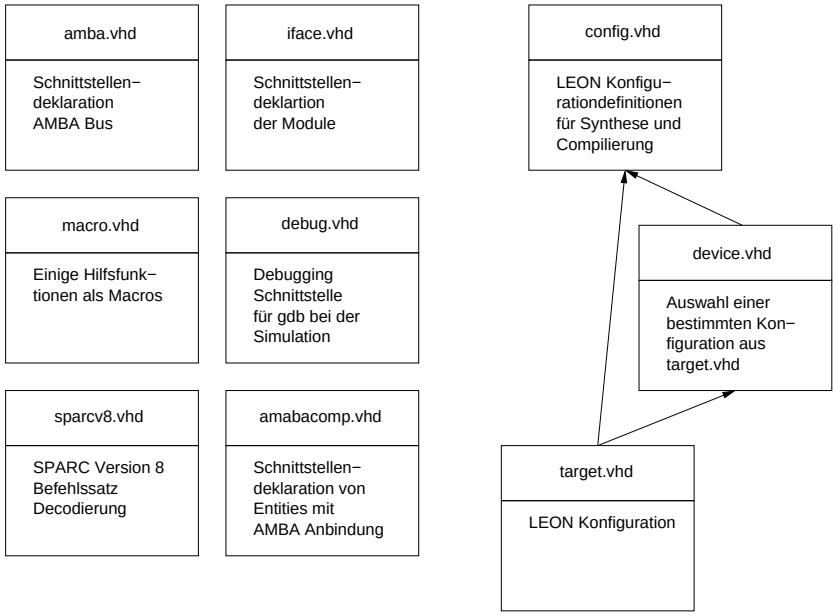

Abbildung 2.2: Übersicht über die Schnittstellen- und Konfigurationsdateien

Die Funktion und die Abhängigkeiten der Quelldateien im Verzeichnis *leon* können den Abb. 2.2 und Abb. 2.3 entnommen werden. Die Dateien in Abb. 2.2 enthalten hauptsächlich Konfigurationseinstellungen und Schnittstellendefinitionen. Da diese mit mehreren Dateien in Abhängigkeit stehen, sind sie in einem eigenen

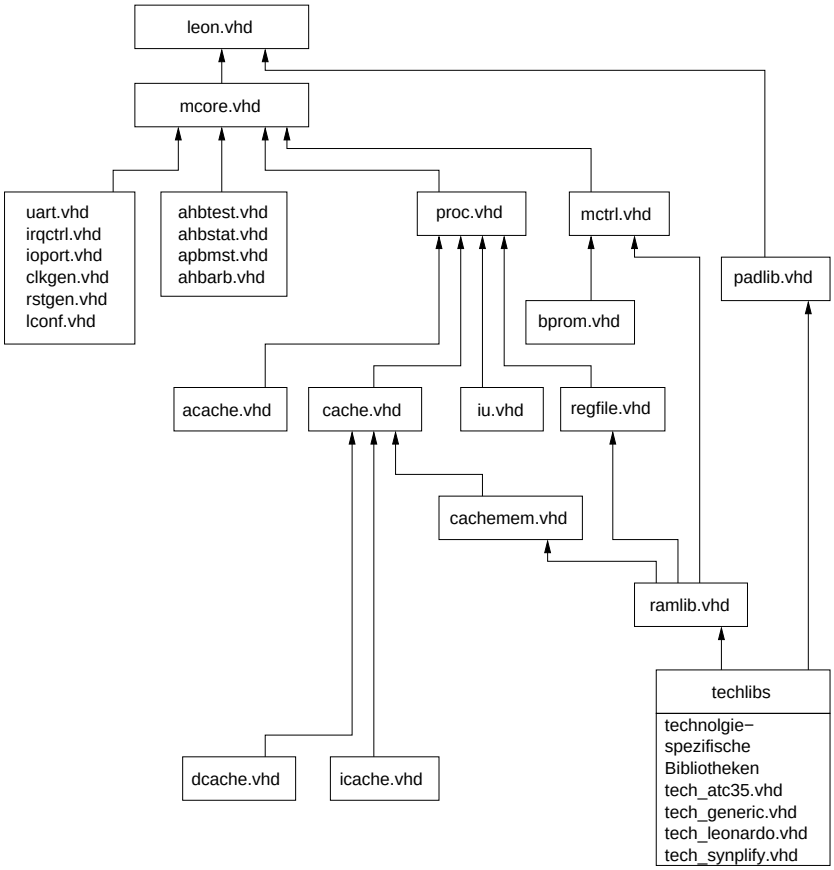

Abbildung 2.3: Übersicht der LEON-Quelldateien

Schaubild untergebracht. Die Datei *target.vhd* verdient von diesen besondere Beachtung, da in ihr alle konfigurierbaren Parameter der LEON-Plattform zu finden und bei Bedarf zu ändern sind. Siehe dazu Abschnitt 2.8.

In Abb.2.3 sind die Abhängigkeiten der restlichen Quelldateien zu erkennen. Die Funktionen, die sich hinter jeder Datei verstecken, lassen sich meistens an deren Namen erkennen. Die etwas schwerer verständlichen sind:

lconf.vhd: LEON Konfigurationsregister; in ihm steht, mit welchen Parametern die CPU synthetisiert wurde

ahbtest.vhd: AHB-Testslave; reagiert mit festen Verhalten

ahbarb.vhd: der AHB Arbiter

apbmst.vhd: die AHB/APB Brücke

ahbstat.vhd: Statusregister des AHB; bei einem Fehler im AHB werden wichtige Daten über Adresse und Daten darin abgelegt

acache.vhd: die AMBA-Schnittstelle für die Caches

2.4 Simulation auf Registertransferebene

Um die Hardware- und Softwareentwürfe zu validieren, gibt es verschiedene Simulationsmodelle. Neben dem reinen Instruktionslevelemulator SIS der in Abschnitt 2.9 erläutert wird, gibt es die Möglichkeit, die Hardwarebeschreibung auf Registertransferebene (RTL) oder auf Gatterebene zu simulieren. Diese Simulationen dienen vor allem zur Überprüfung der Funktionalität der Hardware. Simuliert wurde mit Mentor ModelSIM in der Version 5.4e. In diesem Abschnitt wird die Simulation auf Registertransferebene beschrieben und später, nach der Synthese, die Simulation auf Gatterebene.

Damit die Schaltung simuliert werden kann, muss der Quelltext der Hardware mit einem VHDL-87 kompatiblen Compiler übersetzt werden (siehe Anhang B). Die Schaltung kann für sich alleine nicht simuliert werden. Dazu fehlen noch ein RAM- und ein ROM-Modell und Software, die auf dem System ausgeführt wird. Sie befinden sich in den Verzeichnissen *tbench* und *tsource*. In *tbench* liegt die mitgelieferte Testbench von LEON. Diese kann mit dem darin befindlichen Makefile übersetzt werden. Die Software für die Testbench liegt fertig in den Dateien *ram*.dat* und *rom*.dat* in dem Verzeichnis *tsource*. Das Laden der Software in die RAM- und die ROM-Zellen geschieht automatisch durch die Testbench.

Die Testbench beinhaltet verschiedene Testbenchkonfigurationen, die sich in Einstellungen wie Speichergröße, Speicherzugriffszeit und Betriebsfrequenz unterscheiden. Die vorhandenen Konfigurationen stehen im Verzeichnis *tbench* in der Datei *tbleon.vhd*. Von der Testbenchumgebung wird die Breite der verwendeten ROMs an der parallelen Schnittstelle angelegt. Die Software gibt die einzelnen Etappen und Zustände der Testbench auf dem Datenbus als I/O Ausgabe aus. Die beiden UARTs sind als Nullmodemverbindung miteinander verbunden.

Bei der Simulation auf Register-Transferebene wird der größte Teil des Quelltextes, der auch bei der Synthese verwendet wird, benutzt. Es gibt jedoch für manche Module reine Verhaltensmodelle, die nur bei der Simulation zum Einsatz kommen. Bei der Synthese wird für diese ein anderer Quelltext verwendet, der in der Simulation nicht ausgeführt wird. Dies sind vor allem die RAM-Zellen für Cache und das Registerfile und die Pads des Systems. Um eine möglichst hohe Abdeckung des Quelltextes bei einer Simulation auf Registertransferebene zu haben, gibt es die Möglichkeit, die Technologie abhängigen RAM-Zellen bei der Verhaltenssimulation mit zu simulieren. Es können dazu die Verhaltensmodelle

der Zellen in der Datei *tech_virtex.vhd* verwendet werden. Besser ist aber die Verhaltensmodelle von Xilinx dafür zu verwenden. Diese sind in den Xilinx *unisim* Bibliotheken enthalten. Sollen diese verwendet werden, muss das mitgelieferte Verhaltensmodell der RAM-Zellen in der Datei *tech_virtex.vhd* auskommentiert werden. Damit die Datei dafür nicht ständig geändert werden muss, wurde eine eigene Datei mit dem Namen *tech_virtex_unisim.vhd* für die Simulation mit den *unisim* Bibliotheken angelegt. Das Makefile in *leon* wurde dafür entsprechend angepasst. Mit *make unisim* wird LEON mit den *unisim* Bibliotheken übersetzt. Eine Änderung in der Datei *target.vhd* ist dazu aber noch nötig und wird im Abschnitt 2.8 erklärt.

2.4.1 Inhalt der Testbench

Die mitgelieferte Testbench prüft die Funktionalität der einzelnen Komponenten von LEON. Dazu wird ein spezielles Testprogramm auf der CPU ausgeführt. Der Quelltext des Programms befindet sich in dem Verzeichnis *tsource*. Mit dem Makefile kann, mit dem von der ESA bezogenen C-Compiler, der Quelltext übersetzt werden. Das Ergebnis wird in den Dateien *ram*.dat* und *rom*.dat* gespeichert. In ihnen stehen die Programme als Speicherauszug im Hexdump Format und werden vom Simulationsmodell der Testbench verwendet. Dieses liest deren Inhalt in die simulierten RAM- und ROM-Zellen. Die Erweiterungen der Namen deuten an, für welche Speichermodelle sie benutzt werden. Ein großer Teil der Testbench ist in Assembler geschrieben. Nur damit ist es möglich, alle Funktionen der CPU anzusteuern, spart aber auch Platz. Durch die kompakte Form der Software wurde die Ausführungszeit des Tests kurz gehalten.

Die Testbench besteht auf den folgenden Teilkomponenten, welche in dieser Reihenfolge nacheinander auf der CPU ausgeführt werden:

- Test der Speicherschnittstelle
- Cachetest
- Registerfiletest
- Test des Interrupt Controllers
- Test der Timer, des Watchdog und der power-down Funktion
- Test der parallelen Schnittstelle
- Test der seriellen Schnittstellen

Die Testbench enthält noch Tests für die FPU und den EDAC-Speicher (Error Detection And Correction) der Fehler toleranten LEON-Version. Diese Komponenten sind in der Plattform der Arbeit nicht enthalten und werden nicht getestet. Die Testbench merkt dies automatisch. Bei den Tests der Peripherie muss jedoch darauf geachtet werden, ob diese auch vorhanden sind. So wird z.B. der Watchdog für die FPGAs Designs nicht mitsynthetisiert und ist in der Simulation auf Gatterebene nicht vorhanden. Die Testbench würde unverändert mit einem Fehler beenden. Um die Testbench im Umfang anzupassen, kann die Datei *leon_test.c* in *tsource* bearbeitet werden. Eine Ausführung der Testbench auf realer Hardware ist nicht möglich, da die Auswertung und Ansteuerung der äußeren Signale fehlt.

2.4.2 Simulation von Standardsoftware

Die bisherige Simulation dient vor allem zum funktionalen Test der Plattform. Dazu wird die mitgelieferte Testbench ausgeführt. Für eigene Hardwareentwicklungen ist es jedoch interessant, eigene Programme im normalen Betriebsumfeld auszuführen. Damit lässt sich diese Software in sehr realen Bedingungen testen und bietet vor allem die Möglichkeit, dazugefügte Hardware zu testen. Mit Simulation ist z.B. sehr gut die Validierung, der Kommunikation der LEON-Plattform mit der UDF über den AMBA-Bus, möglich. Um die Simulation unter normalen Betriebsbedingungen auszuführen, wurde die ursprüngliche Testbench abgeändert.

Die Speichermodelle für RAM und ROM wurden übernommen, die Ansteuerung für den Testbenchablauf jedoch entfernt. Die seriellen Schnittstellen sind nicht mehr gegenseitig miteinander verbunden. Es entstehen keine für die Testbench benötigten, Ereignisse von außen an dem System. Das System wird nur mit den RAM- und ROM-Zellen verbunden und bekommt den Takt und das Resetsignal vorgegeben. Damit die CPU andere Programme ausführt, muss der Inhalt der Datei *rom*.dat* in *tsource* geändert werden. Wie dies zu erfolgen hat, wird in Abschnitt 2.9 beschrieben.

Die veränderte Testumgebung befindet sich in dem Verzeichnis *tenv*. In ihr ist, wie auch bei der Testbench, ein Makefile für die Übersetzung vorhanden. In dem Verzeichnis *tenv32_back* befindet sich eine leicht abgeänderte Version der Testumgebung. Diese ist für die Simulation des LEON Designs für das XSV-800 Board auf Gatterebene erforderlich und wird im Kapitel 3 näher beschrieben.

2.5 Synthese mit SynopsysDC

LEON ist als SoC-Plattform darauf ausgelegt mit vielen verschiedenen Syntheseprogrammen synthetisiert zu werden. Für die Programme Synplicity Synplify,

Mentor Leonardo, Synopsys Design Compiler und Synopsys FPGA Compiler werden Skripte oder Projektdateien mitgeliefert. Zudem unterstützt LEON mehrere Zieltechnologien und ist leicht an neue anpassbar. In der Arbeit wird, für die Synthese, SynopsysDC in der Version 1999.10 verwendet. Die Zieltechnologie ist Xilinx Virtex, mit der meisten LEON Entwickler arbeiten.

In der Version 2.1 von LEON war nur Unterstützung für Synplify im Zusammenspiel mit Xilinx vorhanden. Das liegt an der Möglichkeit Synplifys, RAM Zellen automatisch an die Zieltechnologien anzupassen (automatically inferred). Dies bietet SynopsysDC, welches hauptsächlich für ASIC Designs verwendet wird, nicht. In der Datei *ramlib.vhd* im Verzeichnis *leon* sind die Technologie abhängigen Module zu finden. Zur Synthese mit Synopsys, wurde die Datei um die reale Ansteuerung der RAM-Zellen und der Pads erweitert. Zusätzlich musste ein Buildskript für die Verwendung von Virtex mit Synopsys erstellt werden. Am Ende konnte die Funktionalität des Designs mit einer Backannotation validiert werden.

Bei LEON Version 2.2 wurde die Struktur der Dateien geändert. Die technologischen Abhängigkeiten sind in einzelne Dateien verpackt. Die feste Einbindung der RAM-Zellen für Virtex ist verbessert in die Datei *tech_virtex.vhd* übernommen worden. Das Einfügen der Pads in das Design funktioniert ab LEON 2.2 mit Synopsys automatisch, weswegen Padmodelle in der Datei nicht mehr benötigt werden. Dies hat große Vorteile für das später in der Arbeit verwendete Hülldesign (siehe Abschnitt 3.7), für das XSV-800 Board, welches um das LEON-Design gelegt werden musste. Wären dabei die Pads fest eingebunden, müssten diese entfernt und in dem neuen Design eingebaut werden. Bei der LEON Version 2.2 liegt ein Prototyp des Buildskriptes für die Synthese mit SynopsysDC bei und musste an die Zieltechnologie angepasst werden. Dies sind vor allem Einträge für das Einfügen der Pads, dem richtigen Zeitverhalten und dem Laden der richtigen Xilinx Bibliotheken.

Am Ende der Arbeit lagen drei verschiedene Designs für das XSV-800 Board vor. Sie unterscheiden sich in der Speicheransteuerung und im Bootvorgang. Für jedes von ihnen existiert in dem Verzeichnis *syn*, ein entsprechendes Skript. Die Erweiterung der Dateien ist *.dc* (dc-Datei). Mit dem Aufruf von *dc_shell -f <skriptname>* wird ein Syntheselauf gestartet. Am Ende der Synthese steht eine edif-Datei des entsprechenden Designs in dem Verzeichnis.

2.6 Xilinx-Designflow

Nachdem ein Design synthetisiert ist, liegt es in einer Beschreibung auf Gatterebene vor. Die zieltechnologischen Abhängigkeiten sind dabei nur auf ein paar spezielle Zellen begrenzt. Damit es auf einem FPGA betrieben werden kann, muss es

in passende Funktionen für die CLBs und deren Verdrahtung gewandelt werden. Dazu durchläuft es den Xilinx Designflow. Als erstes wird die edif-Datei mit dem Werkzeug *ngdbuild* in ein Xilinx internes Format gewandelt. Dabei werden die für die RAM-Blöcke benötigten Zell-Makros gewandelt. *ngdbuild* stellt auch in dem Design das richtige Zeitverhalten und das Layout der Anschlusspads für das FPGA ein. Die Informationen hierfür stehen in der ucf-Datei. Das *map* Werkzeug wandelt die Gatebeschreibung in die benötigten Funktionen um. Diese werden mit dem 'placer and router' (*par*) auf dem FPGA platziert und miteinander verdrahtet. Mit *bitgen* wird das Design in den vom XChecker Anschluss benötigten seriellen Datenstrom gewandelt. Ein XChecker Anschluss ist die Standardprogrammierschnittstelle für Xilinx FPGAs. Wird das FPGA über ein FlashRAM personaliersiert, müssen die Daten in einem weiteren Schritt mit *promgen* in eine exo-Datei gewandelt werden. Dieses besteht aus S-Records. Für den kompletten Xilinx Designflow kann auch das Werkzeug *dsgnmgr*, das eine grafische Benutzungsoberfläche hat, benutzt werden. Für die kompletten Synthesevorgänge und den Xilinx Designflow gibt es in dem Verzeichnis *syn* Shellskripte für die verschiedenen Designs des XSV-800 Board. Der komplette Designflow ist in Abb. 2.4 zu sehen.

2.7 Simulation auf Gatterebene

Die Xilinx Werkzeuge lassen den Designflow nicht nur in eine Richtung zu. Zur Kontrolle kann das Design an zwei Stellen wieder in eine Beschreibung auf Gatterebene gewandelt werden. Nach dem Wandeln in das Xilinx Format kann es mit *ngd2vhdl* in eine VHDL-Beschreibung auf Gatterebene zurückgewandelt werden. Das Ergebnis kann wiederum mit ModelSIM simuliert werden. Dazu müssen die Simulationsbibliotheken *simprim* von Xilinx verwendet werden. Nach dem 'place and route' gibt es die Möglichkeit, das Design mit *ngdanno* in ein weiteres Format zurückzuwandeln, das ebenfalls mit *ngd2vhdl* in eine VHDL-Beschreibung auf Gatterebene gebracht werden kann. Diese kann, wie die erste, mit ModelSIM simuliert werden. Der Unterschied zwischen den beiden ist, dass bei der Backannotation das Design komplett für das FPGA abgebildet worden ist. Die direkte Umwandlung spiegelt aber nur die Synthese mit SynopsysDC wieder. Bei der Backannotation ist während der Arbeit ein besonderes Verhalten aufgetreten. Das backannotierte Design funktioniert nur, wenn bei der Verwendung von dem Werkzeug *map* die Option *-u* angegeben wird. Die Option hat zur Folge, dass in dem Design keine unbenutzte Logik entfernt werden darf. Es konnte bis zum Schluss der Arbeit nicht geklärt werden, warum das backannotierte Design ohne die Option nicht funktioniert. Auf der Hardware funktioniert es dagegen.

Mit der Simulation auf Gatterebene wird ein synthetisiertes Design getestet

und damit der gesamte Quelltext, der dabei verwendet wird. Jedoch nimmt die Übersichtlichkeit der Struktur des Designs bei der Simulation mit tiefer reichendem Designflow ab. Manche Signale, die zu einem Register gehören, können nicht mehr als zusammenhängende Struktur erkannt werden. Zudem ist die Debugausgabe der UARTs nach der Synthese nicht mehr möglich.

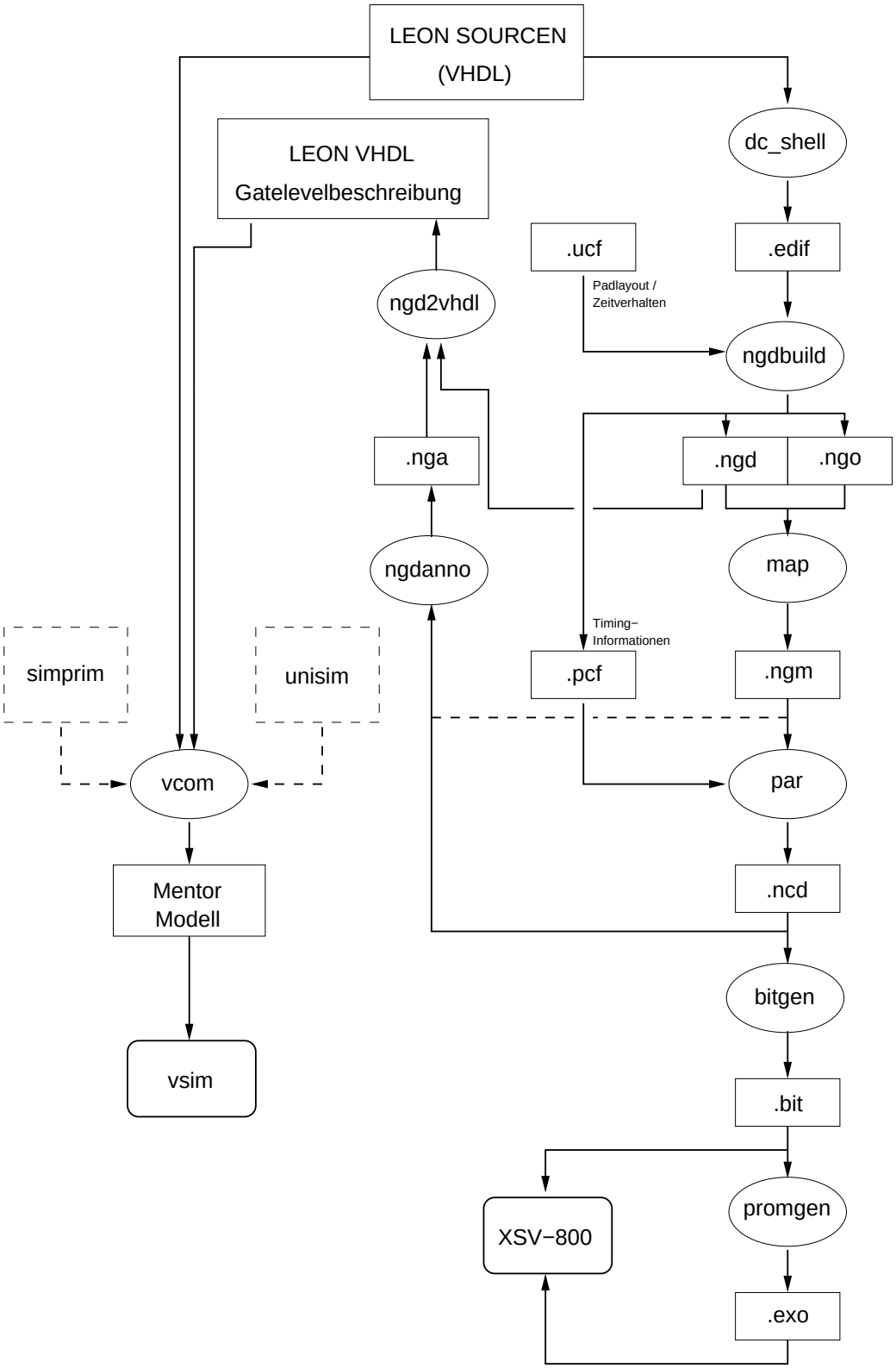

Abbildung 2.4: Designflow der Plattform

2.8 Konfiguration der Plattform

LEON ist eine stark parametrisierbare Plattform. Alle wichtigen Konfigurationsparameter sind in der Datei *target.vhd* zusammengefasst. Diese gibt die enthaltenen Komponenten der Plattform, die Einstellungen der Cachegröße, des AMBA-Bus, die Anzahl der Registerwindos, usw. an. Die Datei ist im Anhang D zu finden. Sie bestimmt was übersetzt oder synthetisiert wird. Damit sie nicht ständig für verschiedene Designs geändert werden muss, gibt es in ihr mehrere Standarddesigns mit vordefinierten Einstellungen. Welches von diesen benutzt wird, steht in der Datei *device.vhd*. Die Datei *config.vhd* wandelt die Einstellungen der *target.vhd* in Parameter mit der Form, wie sie im Quelltext verwendet werden.

Im ersten Teil der *target.vhd* stehen die Typen- und Recorddefinitionen der zu setzenden Parameter. Danach folgen ab Zeile[3] 196 die eigentlichen Parameterzuweisungen.

Als erstes kommen dabei die Synthesewerkzeug abhängigen Einstellungen. Dabei wird für den Parameter *syntool*, der Wert *synplify* für alle Synthesewerkzeuge, außer für Leonardo (*leonardo*), verwendet. Mit *targettech* wird die verwendete Zieltechnologie angegeben, und damit welche *tech_*.vhd* Datei benutzt wird. Für die Synthese mit SynopsysDC muss dieser Parameter auf *virtex* gesetzt sein. Bei der Simulation kann dieser auf *generic*; reines Verhaltensmodell; oder *virtex*; Simulation mit zieltechnologieabhängigem Verhalten; stehen. Die nachfolgenden Einstellungen geben an, ob die Pads, der Speicher, das Registerfile und das BPROM automatisch vom Synthesewerkzeug auf die Zieltechnologie abgebildet werden sollen oder ob die Modelle in den Dateien *tech_*.vhd* dafür verwendet werden. Die restliche Parameter geben an, ob der Systemtakt als gated Clock *(gatedclk*) und das Registerfile synchron für Schreiben und Lesen (*rfsyncrd, rfsyncwr*) in der Schaltung ausgelegt werden sollen. Die Parameter für die Simulation stehen in der Konstanten *syn_none*; für SynopsysDC in *syn_virtex*.

Nach den allgemeinen Syntheseeinstellungen folgen die Konfigurationseinstellungen für LEON. Diese sind in die Bereiche IU, FPU, CP (Coprozessor), Cache, Speicherschnittstelle, Boot, PCI, Peripherie, Debug und AMBA eingeteilt. Die meisten Parameter sprechen mit ihren Namen dabei für sich selbst. Es werden kurz die wichtigsten daraus erklärt.

In dem Bereich Boot werden die verschiedenen Möglichkeiten des Systemstart konfiguriert. Dieser kann durch Setzen der Variable *boot* auf

- *memory*: Boot vom ROM
- *prom*: Boot vom internen BPROM

[3] die Datei ist ab dort im Anhang aufgeführt

- *icache*: Boot aus vorinitialisiertem Cache

ausgewählt werden. Die restlichen Einstellungen sind nur für das interne BPROM relevant.

Im Abschnitt Debug können für die Simulation Debugoutputs eingeschaltet werden. Interessant ist der Parameter *uart*, der ermöglicht, dass die UARTs auf der Standardausgabe ausgegeben werden. So kann unter ModelSIM die Textausgabe der Programme verfolgt werden. Eine Texteingabe ist nicht möglich. Diese ist zwar mit VHDL realisierbar [Ash96], doch ist es problematisch, den Zeitpunkt für die Eingabe zu finden. Die Simulation würde dazu stehen bleiben müssen. Eingaben während der Simulation machen ohnehin wenig Sinn, da diese bei einem realen Takt von ungefähr 75 Herz viel zu langsam läuft, um größere Programme darauf auszuführen.

Im restlichen Teil der Datei *target.vhd*, stehen die zur Auswahl stehenden Synthesemodelle, die mit dem Eintrag in *device.vhd* ausgewählt werden.

2.9 Programmierung und Ausführung

C-Programme können mit dem Crosscompiler wie mit einem gewöhnlichen Compiler übersetzt werden. Der Compileraufruf heißt *sparc-rtems-gcc*. Es gibt dabei jedoch Einschränkungen bei der Verwendung von Bibliotheken, da nur eine standalone C-Bibliothek vorhanden ist. Viele Programme nutzen aber Bibliotheken, die zum Betriebssystem gehören. Die Standardausgabe der Programme wird auf UART1 ausgegeben. Die vom Compiler erzeugten Binaries können mit dem Instruktionslevelemulator SIS emuliert werden. Der Aufruf lautet *sparc-rtems-sis* [Gai00c]. Da der Compiler normalen SPARC-Code erzeugt, können die Programme auch mit dem *gdb* ausgeführt werden. SIS bietet jedoch Einstellungen der Speicherkonfiguration, der Systemfrequenz, der UARTs und einigem mehr. Damit wird das Zeitverhalten kontrollierbar und die Ausgaben des Programms auf die UARTs werden auf der Standardausgabe sichtbar. SIS kann auch vom *gdb[RMS98]* eingebunden werden, womit komplette Funktionalität für das Debugging gegeben ist. Eine genauere Erklärung des Crosscompilers findet sich in [Gai99].

Um die Programme auf der simulierten oder synthetisierten Plattform auszuführen, benötigen diese einen Bootloader. Dabei gibt es drei Varianten, auf welche Weise dieser geladen wird, also drei Methoden, das System zu starten.

Die eine Art ist, den Instruktionscache des LEON vor dem Start mit einem Programm zu initialisieren. Diese Variante gibt es nur für die Xilinx Virtex Technologie. Sie stammt noch aus den älteren LEON Versionen ohne BPROM und wurde in dieser Arbeit nicht benutzt.

Von dem internen BPROM zu starten ist die zweite Möglichkeit. In dem 1 kByte großen ROM steht ein Programm, dass systematisch den Speicher untersucht und danach die Speicherkonfigurationsregister von LEON setzt. Nach dem Initialisieren wartet es auf die Übertragung eines Programms auf der seriellen Schnittstelle. Das Programm muss dazu im S-Record Format vorliegen. Ein vom Compiler erzeugtes Binary kann mit *sparc-rtems-objcopy -O srec –adjust-vma=0x40000000 –set-start=0 <binary> <binary.srec>* in die benötigte Form gebracht werden. Am Ende der erzeugten Datei steht auch eine Startanweisung, die das Programm nach Übertragung startet. S-Records sind ASCII-Strings, die mit einem 'S' beginnen. Gefolgt wird dies von der Typenkennung des S-Records, der Anzahl enthaltener Datenbytes und der Zieladresse im Speicher. Danch kommen die eigentlichen Daten, gefolgt von einer Prüfsumme. Alle Zahlenwerte sind dabei Hexadezimalwerte in ASCII Darstellung.

Als letzte Methode, das System zu starten, wird ein herkömmliches externes ROM verwendet. Da dies viel größer sein kann, als das BPROM, kann die ganze Anwendung in diesem untergebracht werden. Dazu muss das Programm mit einem Bootheader versehen werden. Dieser wird mit dem Werkzeug *mkprom* erzeugt. Die Benutzung des Werkzeugs ist in [Gai00b] erklärt. Bei der Erzeugung der ROM-Daten für die Simulation mit ModelSIM sind dabei folgende Dinge zu beachten. Das ROM sollte die Programmdaten nicht komprimiert enthalten. Dies kann mit der Option *-nocomp* beim Aufruf von *mkprom* verhindert werden. Das so erzeugte ROM kann anschließend mit *sparc-rtems-objdump* und den Optionen *-d* oder *-s*, als die für die Simulation benötigte ROM-Datei als Hexdump oder als Assembler-Code ausgegeben werden. In dem ROM-Code sollte die Abschnitte <memclr> und <_clean> mit *nops* (hexcode: 01000000) deaktiviert werden. An welchen Adressen diese stehen kann in dem Assembler-Output nachgesehen werden. Dieses Vorgehen ist für eine akzeptable Ausführungszeit der Simulation wichtig. Ohne die Änderungen wird erst der komplette Speicher gelöscht. Vor der Übertragung der Software in das RAM ein zweites Mal. Zum Schluss würde die Anwendung zeitaufwendig entpackt. Die Datenbreite des ROM wird beim Systemstart an den beiden unteren Bit (PIO[0:1] der *leon* Entity) der parallelen Schnittstelle angelegt ("00" $\equiv$ 8 Bit; "01" $\equiv$ 16 Bit; "1x" $\equiv$ 32 Bit).

Bei der mitgelieferten Softwareentwicklungsumgebung ist auch das Realtime Betriebssystem RTEMS dabei. Dies wurde nicht verwendet und wird daher nicht weiter diskutiert.

2.10 AMBA

AMBA (Advanced Microcontroller Bus Architecture) ist ein offener Busstandard von ARM. Seit der LEON Version 2.2 wird diese als Systembus verwendet. AM-

BA ist in drei Bussysteme und Protokolle unterteilt. Dies sind der Advanced High-performance Bus (AHB), Advanced System Bus (ASB) und der Advanced Pripheral Bus (APB). AHB ist die Weiterentwicklung des ASB und bietet diesem gegenüber eine höhere Durchsatzleistung. Der APB ist für die Ansteuerung von Peripheriegeräten wie UART, Timer, PIO, Keyboard konzipiert. Er hat eine leicht zu implementierende Schnittstelle und einen geringen Stromverbrauch. Der APB ist über eine Bridge entweder an den AHB oder den ASB angeschlossen. LEON besitzt einen AHB mit einem APB, wie in Abb. 2.1 zu sehen. Im weiteren werden nur diese beiden Bussysteme erklärt. In Abb. 2.5 ist ein vereinfachtes Modell des AMBA-Bus mit AHB und APB zu sehen. Einige Leitungen wurden der besseren Übersicht wegen weggelassen. Dazu gehören die Reset- und Clockleitungen, die es in beiden Bussystemen gibt. In LEON wird der Systemclock und der System-reset für den AMBA-Bus verwendet. Ein einzelner Busreset ist nicht möglich. Die Datenleitungen des AMBA-Busses sind 32 Bit breit.

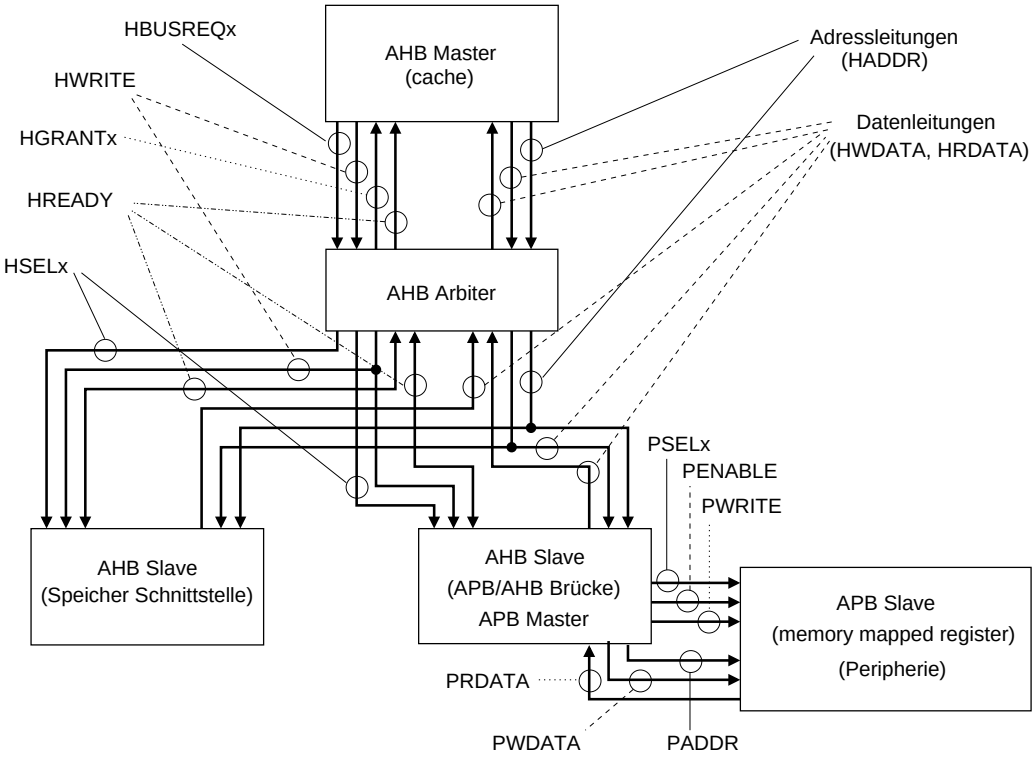

Abbildung 2.5: Vereinfachte Darstellung des AMBA-Bus

2.10.1 APB

Über den APB werden die Speicher abgebildeten Register der Peripherie angesprochen. Dies sind normalerweise Konfigurationsregister, zu denen nur geringer Datendurchsatz benötigt wird. Jedes Peripheriemodul hängt als Slave an dem APB. Sie erhalten über den einzigen Master, der APB/AHB Brücke, Datenzugriffe. Die Brücke ist wiederum ein Slave auf dem AHB. In Abb. 2.5 unten rechts sind die einzelnen Leitungen des APB zu sehen. Der Einfachheit halber ist nur ein APB Slave in der Zeichnung eingezeichnet. Ein weiterer Slave muss, wie der erste, mit allen Leitungen an der Brücke angeschlossen werden. Als Systembus in einem eingebetteten System ist der Bus nicht als Tristate-Bus ausgelegt. Es sind also für jeden Slave eigene Leitungen vorhanden.

Bekommt die Brücke einen Zugriff auf einen ihrer Slaves als Auftrag über den AHB, kann sie über die Adresse decodieren, welcher Slave angesprochen ist. Über die Leitung *PSELx* wird diesem Slave signalisiert, dass auf ihn ein Zugriff erfolgt. Das 'x' in dem Leitungsnamen bedeutet, dass für jeden Slave eine eigene, ihm zugeordnete, Leitung existieren muss. Zeitgleich mit dem Signal *PSELx* wird auch über die Leitungen *PADDR*, *PWDATA* und *PWRITE* die Adresse, die Schreibdaten und die Richtung des Datenzugriffs übergeben. Dieser Zustand wird 'setup' genannt und dauert genau einen Takt auf dem APB. Der nächste Zustand heißt 'enable' und wird durch Setzen der Leitung *PENABLE* auf logisch '1', signalisiert. Dieser Zustand dauert ebenfalls einen Takt. In ihm muss bei einem Lesezugriff der Slave die Daten auf der Leitung *PRDATA* übergeben. Im nächsten Takt ist der Zugriff zu Ende und alle Signale werden gelöscht. Der Zustand 'idle' wird eingenommen.

2.10.2 AHB

Der AHB ist wesentlich komplizierter aufgebaut als der APB. Er ist für hohen Durchsatz konzipiert. Wie der ASB unterstützt er im Unterschied zum APB mehrere Busmaster. Nach AMBA Spezifikation sind bis zu 16 Master möglich. Zusätzlich kann ein AHB Bursttransfers ausführen, bei denen Daten, die im Adressbereich aufeinander folgen, in einem sequentiellen Transfer transportiert werden. Um einen hohen Durchsatz zu erzielen, ist der Bus als Pipline ausgelegt. Dabei werden die Adress- und Steuerdaten eines Transfers einen Takt lang auf den Bus gelegt. Die dazugehörigen Daten werden in den darauffolgenden Takten transportiert. Während dieser Takte werden schon die nächsten Adress- und Steuerdaten angelegt. Zur Handhabung des Busses sind einige Leitungen mehr erforderlich als beim APB, und das Protokoll ist aufwendiger. In dieser Arbeit wurde ein Master für den AHB entwickelt. Er benutzt nur einen einfachen nicht sequentiellen Transfer. Die Erläuterungen zum AHB beziehen sich nur auf die

Grundlagen, um einen Master mit einem solchen Transfer zu entwickeln. Für weitere Details sei auf das Referrenzhandbuch [ARM99] verwiesen.

Der Arbiter ist die Kontrollinstanz, um mehrere Master, die alle Aktionen auf dem Bus auslösen können, zu verwalten. An ihn richten sich die Anfragen der Master, wenn sie den Bus benutzen wollen. Er gibt an, welcher Master den aktuellen Zugriff auf den Bus hat. In Abb. 2.5 sind die wichtigsten Leitungen für einen simplen Transfer zu sehen; der Signalverlauf dazu in Abb. 2.6. Der Master signalisiert über die *HBUSREQx* dem Arbiter, dass er eine Aktion auf dem Bus ausführen möchte. Ob er Zugriff auf den Bus hat, wird dem Master über die *HGRANTx* Leitung gemeldet. Bevor der Master jedoch einen Zugriff auf den Bus machen darf, muss er die Bereitschaft des Slave abwarten, der gerade arbeitet. Die Slaves melden ihre Bereitschaft über die *HREADY* Leitung. Diese ist an allen Mastern und Slaves auf dem Bus angeschlossen. Ein Slave bekommt dadurch auch das Ende der Arbeit eines anderen Slaves mit. Nach dem Takt, bei dem *HGRANTx* und *HREADY* auf ’1’ waren, ist der Master Besitzer der Adress- und Steuerleitungen und legt diese spätestens jetzt auf dem Bus an. Diese Phase dauert, wie oben erwähnt, genau einen Takt lang. Will der Master noch weitere Daten transportieren, kann er die Adresse des nächsten Transfers im nachfolgenden Takt anlegen. Bei einem einzelnen Zugriff muss er die Kontrolle über den Bus durch Löschen der *HBUSREQx* Leitung abgeben. Ein erneutes *HREADY* von den Slaves signalisiert, dass nun auch die Datenleitungen für den Transport bereit stehen. Im nächsten Takt muss der Master seine Daten auf den Bus legen. Mit einem erneuten *HREADY* Signal signalisiert der Slave, dass die Daten eines Lesezugriffs an dem Bus anliegen. Ist es ein Schreibzugriff, singlisiert er damit, dass die Daten übernommen wurden. In Abb. 2.7 ist derselbe Transfer mit optimalem Zeitverhalten zum Vergleich zu sehen.

Die Signale werden beim AHB jeweils in der positiven Flanke des Bustakts angelegt und liegen mindestens bis nach Beginn der nächsten positiven Flanke an. In den Diagrammen wird zwischen Signalen, die auf einer, und denen, die auf mehreren Leitungen, übertragen werden, unterschieden. Bei den Signalen auf einer Leitung wurde ein grau eingefärbter Übergangsbereich zur übersichtlicheren Darstellung eingezeichnet. Ist das Signal HREADY als logisch ’0’ und ’1’ eingezeichnet, kann einer der beiden Werte anliegen, was von der vorhergehenden Aktion auf dem Bus abhängt. Bei Signalen auf meheren Leitungen kann keine genaue Zuordnung zwischen ’1’ oder ’0’ gemacht werden, weswegen nur die Übergangszeitpunkte mit einem kurzen Zusammenlaufen der Signale markiert sind. Der beschriftete Bereich gibt dabei an, wann die Daten für den Transfer auf dem Bus anliegen müssen.

Der AHB unterstützt im Gegensatz zum ASB Bursttransfers. Für deren Steuerung sind einige Leitungen nötig, die bisher unter dem Namen Control zusammengefasst wurden. Es folgt eine kurze Erklärung, mit erforderlicher Belegung

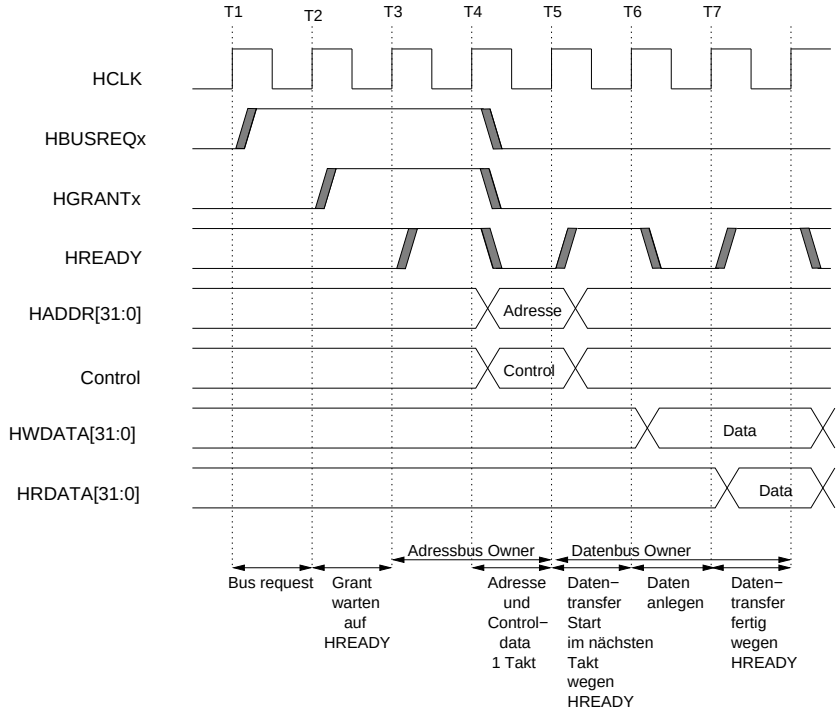

Abbildung 2.6: Simpler Datentransfer auf AHB

für einen einzelnen Transfer in eckingen Klammern.

HTRANS[1:0] Gibt den Zustand und die Art der Datenübertragung an (IDLE, BUSY, NONSEQ, SEQ). Es kann passieren, dass der Arbiter dem Master ein GRANT signalisiert, obwohl dieser keinen REQUEST gestartet hat. In diesem Fall muss der Master nach AHB-Spezifikation mit IDLE antworten. [NONSEQ]

HBURST[2:0] Gibt die Art des Burst an (SINGLE, INCR, WRAP4, INCR4, WRAP8, INCR8, WRAP16, INCR16). [SINGLE]

HSIZE[2:0] Gibt die Datenlänge des Transfers an (8, 16, 32 (Word), 64, 128, 256, 512, 1024 Bits). [Word]

Bei Zugriffsfehlern, Zugriffsverzögerungen oder sonstigen Ereignissen gibt der Slave entsprechende Meldung über die Leitung *HRESP[1:0]*. Normalerweise legt der Slave bei erfolgreichem Zugriff ein 'OKAY' darauf an. In dem einzelnen Zugriff des hier verwendeten Masters wird dieses Signal nicht ausgewertet. Bei einem Fehlerfall muss der Master den Zustand 'IDLE' auf der *HTRANS* Leitung anlegen. Dieser wird von der Schaltung aber ohnehin bei nicht aktivem Master

auf der Leitung angelegt. Dadurch ist auch die korrekte Antwort bei einem Grant bei nicht angefordertem Bus garantiert.

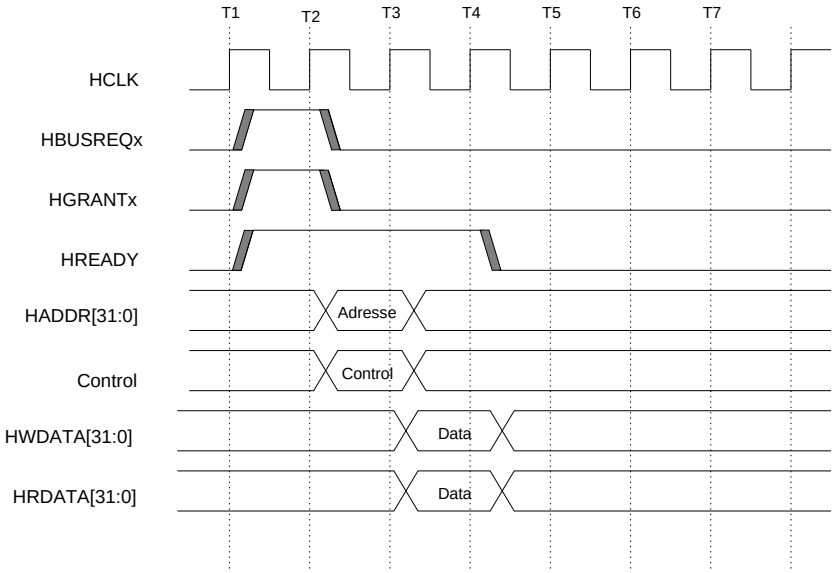

Abbildung 2.7: Simpler Datentransfer mit optimalem Zeitverhalten

Kapitel 3

Prototyping von LEON mit XSV-800 Board

Für die Entwicklungsumgebung wurde ein Evaluierungsboard beschafft, um die erzeugten Schaltungen in Echtzeit testen zu können. Dazu wurde ein zur Aufgabe passendes FPGA basiertes Entwicklungsboard aus der auf dem Markt erhältlichen Hardware ausgesucht. Das Board besitzt einen FPGA vom Typ Virtex der Firma Xilinx [xil01]. Dadurch können die in der Abteilung schon vorhandenen Entwicklungswerkzeuge von Xilinx verwendet werden. Das Board kann als alleinstehendes System betrieben werden. Im Gegensatz zu PCI basierten Rechnereinstecksystemen bringt dies die Unabhängigkeit und eigenständige Funktionsfähigkeit des darauf zu implementierenden SoC zum Ausdruck. Es wurde das XSV-800 Board [XES00] der Firma X Engineering Software System Corp. (XESS)[xes01] ausgewählt, da dies eine Menge Standardschnittstellen und im Verhältnis zu vergleichbaren Boards den größten Speicher besitzt. Außerdem arbeiten einige Entwickler, die den LEON implementieren, mit diesem und es war vom Preis/Leistungsverhältnis eines der besten.

3.1 Funktionalität des Boards

Das Board besteht aus einer 152mmX152mm großen Platine. In der Mitte befindet sich das Herzstück, das Xilinx Virtex FPGA XCV800. Auf dieses werden die synthetisierten Schaltungen abgebildet. Über ein Xilinx XC95108 CPLD wird das Board konfiguriert. Die XESS Konfigurationssoftware spricht das CPLD direkt an. Auf dem Board befinden sich 2MB SRAM, aufgeteilt in 2 unabhängige Bänke mit jeweils 16 Bit Datenbreite. Dabei besteht eine Bank jeweils aus zwei 8 Bit SRAM Bausteinen. Zusätzlich zu diesem flüchtigen Speicher gibt es noch einen 8 Bit 2MB FlashRAM-Baustein. Dieser kann zur Speicherung von FPGA-

Personalisierungsdatenströmen, aber auch zur generellen Benutzung, z.B. als SystemROM, benutzt werden. Zur Stromversorgung stehen eine 9VC-Buchse oder eine ATX-Steckverbindung zur Verfügung.

Das Board besitzt außerdem, wie oben erwähnt, viele Standardschnittstellen. Für die meisten von ihnen befindet sich zusätzlich noch ein jeweils dazu passender IC mit auf dem Board. Die ICs werden zur Umwandlung von analogen Signale in digitale und umgekehrt benötigt, da das FPGA ein rein digitaler Baustein ist. Außerdem verhindern diese ICs, dass die für die Benutzung der Schnittstellen nötigen Schaltungen immer mit auf das FPGA abgebildet werden. Auf dem Board sind folgende Schnittstellen vorhanden:

- ein Videodecoder, der NTSC/PAL/SECAM Signale digitalisiert mit einem S-VHS und einem Composite Eingang
- ein RAMDAC, der die Ausgabe von Videodaten auf eine VGA-Buchse ermöglicht
- ein Digital/Analog (D/A) und Analog/Digital (A/D) Konverter, der Stereo Audiosignale bearbeiten kann; er besitzt zwei Stereo 1,5 mm Klinkenbuchsen
- ein Ethernet-Adapter, der 10/100Mbps-Signale bearbeiten kann mit einer Standard RJ45-Buchse
- eine PS/2-Buchse
- eine serielle Schnittstelle mit einem 9 Pin D-Sub-Stecker
- eine parallele Schnittstelle mit einer 25 Pin D-Sub-Buchse; sie wird standardmäßig von XESS zur Programmierung des Boards verwendet
- ein USB-Anschluss
- Ein XChecker-Anschluss; der Standardanschluss von Xilinx, um FPGAs zu programmieren und zu steuern; mit diesem kann das CPLD auf dem Board nicht angesprochen werden

Zusätzlich gibt es auf dem Board noch zwei 7-Segmentanzeigen und eine zehnstellige LED-Strichanzeige, mit denen Statusinformationen ausgegeben werden können. Dazu kommen noch vier Mikrotaster und ein achtstufiger DIP Schalter und schließlich noch zwei allgemein benutzbare Erweiterungsschnittstellen mit insgesamt 76 Leitungen zum FPGA, die sich aber dieselben Leitungen wie das SRAM zum FPGA teilen. Dass sich manche Komponenten die Leitungen mit anderen zum FPGA teilen müssen, kommt auf dem Board mehrfach vor, was an der begrenzten Anzahl von Anschlüsse am FPGA liegt.

3.2 PLDs

Um größere Schaltungen auf einem Chip unterzubringen, ohne ASICs zu verwenden, gibt es die programmierbaren Bausteine (PLD, programmable logic device). Beim klassischen PLD kann dabei eine in disjunktiver Normalform vorliegende Funktion in den Chip programmiert werden. Dabei laufen die Eingangssignale über programmierbare Verbindungsstellen zuerst durch eine UND-Matrix. Das Ergebnis der UND-Matrix wird danach entweder ODER-Verknüpft oder an eine programmierbare ODER-Matrix weitergeleitet, wobei die Ausgabesignale entstehen. Wie genau die ODER-Verarbeitung aussieht, unterscheidet die Bausteine in programmable array logic (PAL), programmable logic array (PLA) und generic array logic (GAL). Bei letzterem kann, durch Zurücksteuern der Ausgänge auf die Eingänge und Hinzufügen einer Speicherzelle an den Ausgängen, mehrstufige und sequentielle Logik umgesetzt werden. PALs und PLAs sind auf zweistufige Logik begrenzt.

Um weitaus komplexere Schaltungen auf einem Chip zu programmieren, gibt es Complex Programmable Logic Devices (CPLD) und die Field Programmable Gate Arrays (FPGA)[Jen94]. Beide beruhen auf den sogenannten Configurable Logic Blocks (CLBs). Diese Blöcke können, vergleichbar mit den klassischen PLDs, logische Funktionen mit einer geringen Anzahl von Eingangsvariablen abbilden, enthalten aber auch Speicherzellen für sequentielle Logik. Der Unterschied zwischen CPLD und FPGA besteht dabei in der Komplexität. Ein CPLD ist weitaus einfacher in der Struktur als ein FPGA. FPGAs besitzen eine große Anzahl von CLBs, die auf ihnen in einer Matrix; Feld; angeordnet sind. Bei CPLDs sind dies weitaus weniger und nur ein-Dimensional angeordnet. Die Verbindungsleitungen zwischen den CLBs können beim FPGA als auch beim CPLD frei programmiert werden.

Alle diese Bausteine gibt es in unterschiedlichen Technologien, die sich in der Art der Personalisierung unterscheiden. Bekannte Techniken sind SRAM, EPROM, EEPROM und Antifuse. Diese sind, wie in der Speichertechnik, für die verschiedenen Eigenschaften in der Reprogrammierbarkeit und Flüchtigkeit der Konfigurationsdaten verantwortlich.

Der genaue Aufbau eines CLB und die Integration zusätzlicher Blöcke, z.B. von RAM, und deren Anzahl unterscheiden sich zwischen den verschiedenen FPGA-Modellen.

3.3 FPGA

Auf dem Board befindet sich das Xilinx Virtex XCV800 FPGA mit Speedgrade 4. Der Speedgrade gibt die Verzögerungszeiten der Blöcke auf dem Baustein

an. Je größer der Wert ist, desto schneller schalten die Blöcke. Der Speedgrade wächst ungefähr linear mit der Schaltgeschwindigkeit der Bausteine. Das FPGA auf dem Board kann maximal 888 439 Systemgatter abbilden. Es besteht aus einer 56x84 Configurable Logic Block (CLB) Matrix. Jeder CLB besitzt 2 Slices. Eine Slice besteht wiederum aus 2 Logic Cells (LC). Somit hat das FPGA $56 * 84 * 2 * 2 = 18816$ LCs. Xilinx gibt hier 21168 an. Der Unterschied entsteht, da Xilinx die Anzahl der CLBs mal 4.5 rechnet, da sich in den CLBs noch weitere Logik befindet und sich damit die Anzahl der abbildbaren Logik erhöht. Jeder LC kann eine logische Funktion mit 4 Eingangsparametern berechnen. Zusätzlich ist noch Carry-Logik und ein Speicherelement darin enthalten. Das FPGA besitzt außerdem 28 RAM-Blöcke auf denen 14 kByte an Daten gespeichert werden können. Bei diesen ist die Datenbreite und die Anzahl der Ports verschieden konfigurierbar. Um die Speicherblöcke in LEON zu nutzen, wird die Datei *tech_virtex.vhd* zur Synthese mit Synopsys benötigt. Die Ansteuerung der RAM-Blöcke steht in [Xil00b].

Das XCV800 ist in der Gehäuseform HQ240 auf dem Board montiert. Diese besitzt 166 I/O Pins. Das genaue Pinlayout und eine genauere Spezifikation des FPGA kann in [Xil00c] gefunden werden. Die Zuordnung der Pins mit den XSV-800 Board Leitungen steht in [XES00].

3.4 CPLD

Neben dem XCV800 FPGA befindet sich ein Xilinx XC95108 CPLD [Xil98] auf dem Board. Dieses übernimmt die Aufgabe, das FPGA zu personalisieren. Das FPGA kann entweder über die Leitungen des XCheckeranschlusses oder über Daten aus dem FlashRAM konfiguriert werden. Mit dem CPLD ist beides möglich, da die Leitungen des extern zugänglichen XCheckeranschlusses auch an ihm angeschlossen sind. Die Möglichkeiten der Programmierung und der Betrieb des Boards wird in den nachfolgenden Abschnitten erklärt.

Das CPLD besitzt sechs Blöcke mit jeweils 36 Eingangssignalen. In ihnen wird von 18 Makroblöcken aus den Eingaben jeweils ein Ausgabesignal erzeugt. Die Ausgaben können wiederum als Eingabe für andere Blöcke dienen oder aus dem Chip ausgegeben werden. Somit können maximal $6*18 = 108$ Pins angesteuert werden. Das TQ100 Gehäuse auf dem XSV-800 Board besitzt aber nur 100. Im Unterschied zum FPGA besitzt das CPLD ein eigenes internes FlashRAM. Seine Konfiguration ist bei Unterbrechung der Stromzufuhr dadurch nicht flüchtig. Deswegen ist bei dessen Programmierung Vorsicht geboten, da auch die zu dessen Konfiguration notwendigen Leitungen in einem Layout benutzt werden können. Sind diese für die Programmierung nicht mehr zugänglich, kann der Baustein nicht mehr benutzt werden.

3.5 Kommunikation und Konfiguration

Zentraler Punkt des XSV-800 Board ist das FPGA. Von ihm gehen Leitungen zu fast allen anderen Komponenten und steuert die Perpheriebausteine der Schnittstellen an. Im Zusammenspiel mit der Programmierung und dem CPLD gibt es jedoch Besonderheiten, die in diesem Abschnitt beschrieben werden.

Die Personalisierung des FPGA kann über den XCheckeranschluss erfolgen. Über es ist mit der von Xilinx entwickelten Software die Programmierung möglich. Xess liefert mit dem Board Software, die einen anderen Weg geht. Hauptverbindung vom Board zu einem externen Rechner ist die parallele Schnittstelle des Boards. Über diesen sind mit der Xess Software alle Möglichkeiten, das Board zu konfigurieren, gegeben. Dazu ist die parallele und die serielle Schnittstelle nicht an das FPGA sondern an das CPLD angeschlossen. Eine direkte Ansteuerung dieser Schnittstelle ist vom FPGA aus nicht möglich. Ein großer Teil der Leitungen von der parallelen Schnittstelle sind mit den Konfigurationsleitungen des CPLD verbunden. Damit kann über entsprechende Software das CPLD personalisiert werden. Um das FPGA, das FlashRAM oder den Taktgeber des Boards zu programmieren, wird jeweils ein entsprechendes Design in das CPLD geladen. Die Leitungen, die vom CPLD zum FPGA gehen, sind daher auch an diesen Komponenten angeschlossen. Zusätzlich ist dies noch bei den Leitungen zum DIP-Switch, zu einem Mikrotaster, zum XCheckeranschluss und zu allen Leuchtdioden der Fall. Die Leitungen zu den Leuchtdioden sind ein großer Teil derer, die auch mit dem FlashRAM verbunden sind. Diese Leitungen sind mit vier Komponenten verbunden. Dies bringt Vor- und Nachteile mit sich. Statusinformationen können vom FPGA und CPLD ausgegeben werden, und bei FlashRAM-Zugriffen geschieht dies automatisch. CPLD und FPGA können beide das FlashRAM benutzen. Die Ansteuerung der 7-Segmentanzeige, vom FPGA aus, ist nur erschwert möglich. Es sind fast keine Leitungen mit freier Verfügbarkeit zwischen FPGA und CPLD vorhanden.

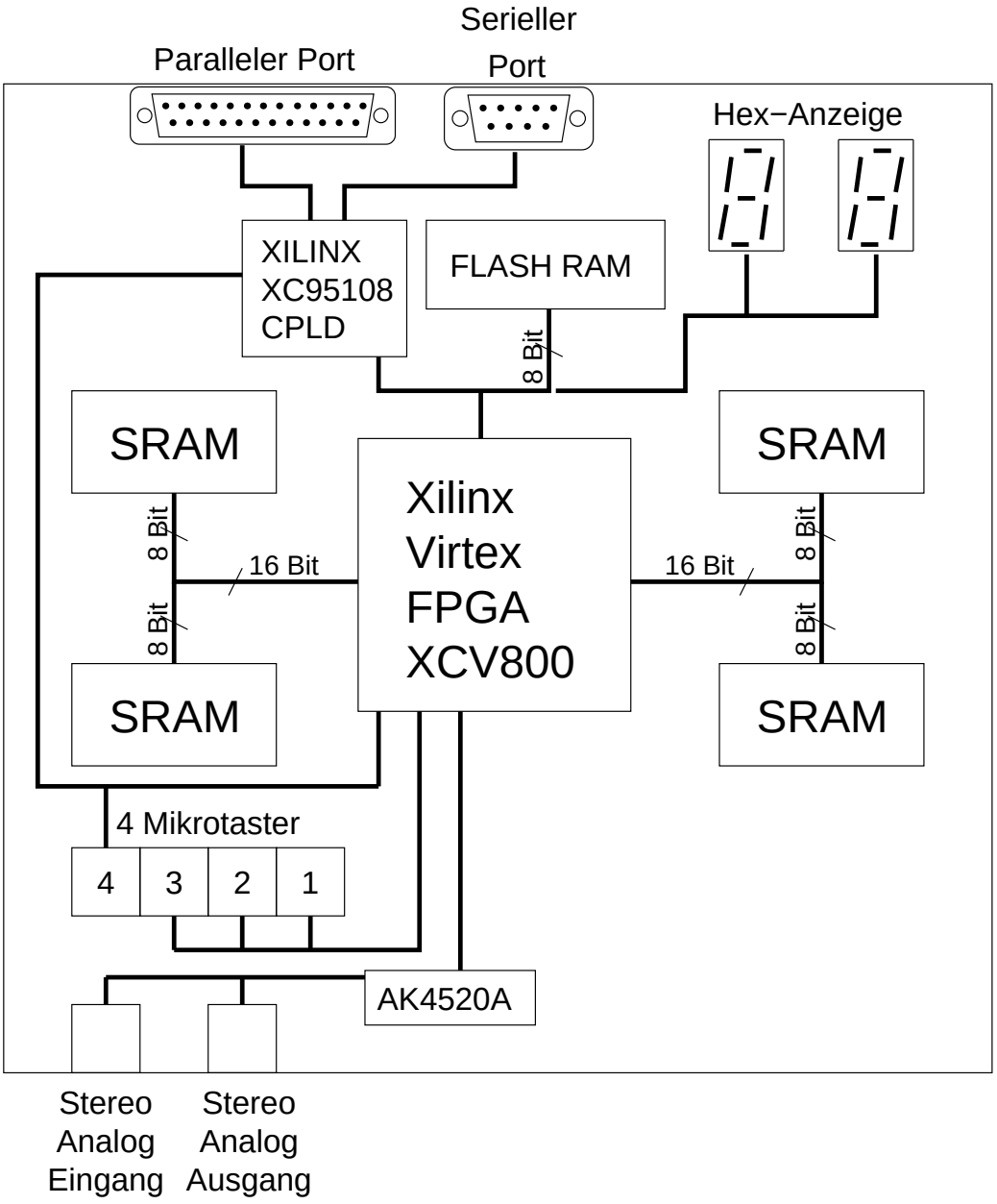

Abbildung 3.1: XSV-800 Boardübersicht

Die Programmierung des FPGA kann auf zwei Arten erfolgen. Auf das CPLD wird ein Design geladen, über welches die Leitungen des XCheckeranschluss auf die parallele Schnittstelle durchgeschliffen werden. Mit der Software von Xess kann so, mit einem für den XCheckeranschluss generierten Datenstrom, das FPGA personalisiert werden. Die andere Möglichkeit ist, über das CPLD Zugriff auf das FlashRAM zu bekommen. Dieses kann mit der Xess Software mit beliebigen Daten gefüllt werden. Mit den Xilinx Werkzeugen können die Personalisierungsdaten des FPGA in eine, für ROM-Programmierung nötige Form gebracht werden. Diese werden in das FlashRAM geladen. Das CPLD wird anschließend mit einem Design programmiert, das das FPGA mit den Daten aus dem FlashRAM personalisiert. Ein so programmiertes Board kann bei Stromzufuhr sich selbst konfigurieren und damit standalone betrieben werden. In Abb. 3.1 ist eine Übersicht über das Board zu sehen. Sie ist stark vereinfacht und enthält nur die Komponenten, die in der Arbeit verwendet wurden.

3.6 CPLD Synthese

Bei der mitgelieferten Software des XSV-800 Board waren die drei nötigen Designs für das CPLD, um das FPGA zu programmieren, enthalten. Diese emöglichen aber nicht die benötigte Kommunikation der LEON-Designs auf dem FPGA über die serielle Schnittstelle. Die Schnittstelle ist nur an dem CPLD angeschlossen und ist in den mitgelieferten Designs nicht durch das CPLD durchgeführt. Deswegen wurden die ursprünglichen Designs von Xess, welche in [Bou00b] und [Bou00a] beschrieben werden, geändert. Die Änderung sind nur bei den zweien, die den FPGA personalisieren, von den dreien nötig, da bei der Programmierung des FlashRAMs keine Verbindung vom FPGA zur seriellen Schnittstelle benötigt wird. Für den Zweck der Änderung an dem Design, welches das FPGA direkt über die parallele Schnittstelle programmiert, konnte auf eine fertige Arbeit im Internet zurückgegriffen werden. Sie musste noch synthetisiert werden. Der Designflow für das CPLD unterscheidet sich zu dem vom FPGA. SynopsysDC benötigt lediglich andere Bibliotheken. Nach Benutzung des Xilinx Werkzeugs *ngdbuild* läuft der Designflow aber einen komplett anderen Weg. Es werden dazu die Xilinx Werkzeuge *hitop*, *hprep6* und *jtagprog* benötigt. Für beide Designs sind Buildskripte in dem Verzeichnis *cpld* vorhanden. Die Synthese ist ebenso mit dem Werkzeug *dsgnmgr* möglich. Der genaue Designflow ist in dem mitgelieferten Handbuch des Xilinx Alliance Paket [Xil00a] zu finden.

3.7 Synthese von LEON für XSV-800

Für das XSV-800 Board wurden drei verschiedene Designs als Grundplattform erstellt. Zwei davon benutzen das interne BPROM von LEON. Sie unterscheiden sich in der Speicherbreite. Das erste benutzt lediglich eine 16 Bit Speicherbank des Boards. Diese Konfiguration war ohne Änderungen des LEON-Designs möglich. Das zweite unterscheidet sich von diesem durch ein Hülldesign, das LEON auf 32 Bit Speicherbreite für die Benutzung beider Speicherbänke erweitert. Das dritte Design benutzt zum Booten das FlashRAM auf dem Board.

LEON bietet verschiedene Konfiguration der Datenbreite an. An der Anzahl der Anschlüsse nach außen ändert sich dabei nichts. Es werden einfach nicht alle Leitungen benutzt. Die Top-Entity von LEON ist im Anhang D.2 aufgeführt. Diese besitzt einen 32 Bit Datenbus, 28 Adressleitungen und die Unterstützung von vier RAM-Bänken und zwei ROM-Bänken. Bei Benutzung eines ASICs würden auf der für das Gesamtsystem entworfenen Platine die RAM- und ROM-Chips entsprechend mit den Bussen verbunden. Bei einem FPGA-System, bei dem das Platinenlayout schon vorher festliegt, ergeben sich damit aber Probleme. Das XSV-800 Board besitzt zwei getrennte RAM-Bänke und ein ROM; FlashRAM. Alle werden mit getrennten Leitungen vom FPGA angesteuert. Um das RAM und das ROM zu benutzen, müssen die Leitungen des Daten- und Adressbus nach außen verdoppelt werden. Für den bidirektionalen Datenbus ist das nicht so trivial. Für die Adressleitungen ist dies auch nicht mit entsprechender Konfiguration der ucf-Datei getan, da dies keine doppelte Zuordnung von Ausgangsleitungen zulässt. Ohne Änderung des LEON-Designs ist somit der Anschluss mehrerer Speicherchips mit dem Board nicht möglich.

Das erste Desgin erlaubt es jedoch, LEON ohne Änderungen auf dem XSV-800 Board zu betreiben. Mit Benutzung des internen BPROM, kann auf das externe ROM verzichtet werden. Benutzt man die 16 Bit Speicherschnittstelle reicht eine Speicherbank des Boards für den Betrieb aus. Der Adressbus wird auf diese Weise nur einmal benötigt und der Datenbus wird nur auf den oberen 16 Bit von LEON benutzt. In dem Verzeichnis *syn* sind die benötigten Dateien für die Synthese zu finden. Das bash-Skript *build_script_bprom_16b* synthetisiert und bildet das Design, für das FPGA, ab. Die Datei *build_xsv800_bprom_16b.dc* ist das dazugehörige Synopsys-Skript. Das Padlayout für den FPGA ist in *xsv800_16b.ucf* zu finden. Dieses legt das Layout für die im Betrieb benötigten Leitungen fest. Alle nicht benötigten Anschlüsse der LEON Top-Entity werden zufällig mit den Pads des FPGA verbunden. Diese werden zwar von dem Synopsys-Skript nicht als Pads eingefügt, geschieht von den Xilinx Werkzeugen nachträglich. In der ucf-Datei gibt keine Möglichkeit Pads explizit nicht anschließen zu lassen. Welche Auswirkungen dies genau auf das Board, den FPGA und die Funktionalität des Designs hat, kann nicht gesagt werden. In der Zeit dieser Arbeit sind beim Be-

trieb keine Auswirkungen aufgetreten, und das Verhalten wurde deswegen nicht weiter untersucht.

Für das zweite und dritte Design wurde die Top-Entity von LEON verändert und eine Hülle darum gelegt. In dem zweiten Desgin wurden beide Speicherbänke des Boards benutzt und somit die Datenbreite auf 32 Bit erhöht. Dazu wird der Adressbus in dem Hülldesign doppelt nach außen geführt. Die Chipselect, Writeenable und Outputenable Leitungen beider Bänke sind dazu mit den entsprechenden Leitungen der Bank 0 von LEON angeschlossen. Wegen der Writeenable Leitung musste zum ersten mal die *leon* Entity verändert werden. Das Signal ist bidirektional ausgelegt. Der Grund dafür ist nicht genau bekannt. Dies könnte im Zusammenhang mit der Fehler toleranten Version von LEON und deren Benutzung in Satellitensystemen stehen. Da auf eine Eingangsleitung nicht zwei Signale gegeben werden können, musste das Signal in eine unidirektionale Leitung geändert werden. Alle bidirektionalen Leitungen der *leon* Entity kommen als unidirektionale Leitungen aus den von ihr eingebundenen Modulen an. Erst in der *leon* Entity werden diese mit Tri-state Pads in bidirektionale Leitungen gewandelt. Für die Writeenable Leitung wurden diese Pads entfernt. Das Ausgangssignal wird direkt in die Eingänge zurück geleitet und zusätzlich nach außen weiter gegeben. Die nicht benötigte Eingangsrichtung von außerhalb wurde damit entfernt. In dem Hülldesign kann auf diese Weise das Writeenable Signal der LEON-Bank 0 auf beide Speicherbänke des XSV-800 Boards gegeben werden. Die geänderte *leon* Entity steht in einer eigenen Datei *leon32.vhd*. Das Hülldesign mit BPROM steht in *xsv800_bprom_32b.vhd*. Damit für das dritte Hülldesign nicht eine eigene *leon* Entity benötigt wird, wurde die Hülle für das 32 Bit Design mit BPROM später an die gleiche *leon* Entity, die für die Flash Ansteuerung benutzt wird, angepasst.

Das dritte Design für das Board beinhaltet die Ansteuerung des FlashRAM als ROM für den Systemstart. Dazu musste der Datenbus verdoppelt werden. Im Gegensatz zur Writeenable Leitung musste dabei die bidirektionale Verbindung bestehen bleiben. In der *leon* Entity wurden dazu alle Signale der Datenleitung, die von den eingebundenen Modulen kamen, nach außen gelegt. Dazu gehört auch die Select Leitung, die die Richtung auf dem Bus angibt. In dem Hülldesign *xsv800_32b.vhd* wurde zur Ansteuerung des Datenbusses ein spezielle Entity mit Namen *selector* geschaffen. Diese generiert die Pads für jeweils 8 Bit des Busses und entscheidet, ob die interne Datenleitung mit der RAM- oder ROM-Leitung verbunden wird und ob dies Eingabe- oder Ausgabesignale sind. Um nicht die Designdaten aus dem FlashRAM beim Systemstart zu lesen, wird die oberste ROM-Adressleitung auf logisch ’1’ gesetzt. Damit kann das ROM-Programm in dem oberen MB des FlashRAMs stehen. Da das Design mehrere Adress- und Datenbusse nach außen führt, kann es nicht mit der normalen Simulationsumgebung *tenv* simuliert werden. Eine speziell für dieses Design angepasste Simulationsum-

gebung steht in *tenv23_back*. Die RAM Größe ist bei dieser auf 1 MB begrenzt, da die Verhaltensmodelle der RAMs nicht mehr zulassen. Bei der derzeitigen Simulationsgeschwindigkeit macht ein größeres RAM aber auch keinen Sinn.

3.8 Betrieb des XSV-800 Board

Die Ansteuerung zur Programmierung und dem Betrieb des Boards erfolgt von einem externen Rechner aus. In der Abteilung wird dazu ein PC mit Linux benutzt. Da die Software von Xess nur für Microsoft Windows mitgeliefert wird, aber der Quelltext erhältlich ist, konnte diese nach Linux portiert werden. Weiteres dazu ist im Anhang C zu finden.

Die Software besteht aus zwei Programmen; *xsload* und *xsport*. Beide sind von der Kommandozeile aus aufzurufen. *xsport* erwartet als Parameter einen 8 Bit Binärwert, in den ASCII-Zeichen ’0’ oder ’1’. Der Wert wird nach Ausführung des Programms konstant auf der parallelen Schnittstelle ausgegeben. *xsport* wurde in der Arbeit nicht benötigt. Beide Programme benötigen zur Ausführung die Umgebungsvariable *XSTOOLS_BIN_DIR* auf das Xess Arbeitsverzeichnis gesetzt. In ihm stehen die von Xess mitgelieferten Designs für CPLD und FPGA.

Mit dem Programm *xsload* kann das XSV-800 Board programmiert werden. Dazu wird dem Programm beim Aufruf ein Dateiname übergeben. In der Datei stehen die Daten, die entweder für das CPLD, das FlashRAM oder das FPGA gedacht sind. Um welchen Datentyp es sich handelt, wird an der Dateiendung erkannt. Dateien mit der Endung .svf (svf-Dateien) sind Designfiles für das CPLD. Die bit-Dateien sind Konfigurationsströme für das FPGA. Der Inhalt von exo-Dateien wird in das FlashRAM geschrieben. Im Nachfolgenden wird die Konfiguration und der Betrieb mit den verschiedenen Designs beschrieben.

Damit auf dem FPGA die LEON-Plattform betrieben werden kann, muss als erstes die richtige Taktfrequenz auf dem Board programmiert werden. Die richtige Jumperstellung dafür ist im Benutzerhandbuch [XES00] von Xess dokumentiert. Auf dem Board befindet sich ein Taktgenerator vom Typ Dallas DS1075. Dieser hat einen maximalen Takt von 100 MHz. Um die Taktfrequenz herab zu setzen, können die 100 MHz durch einen programmierbaren ganzen Faktor geteilt werden. Die LEON-Designs der Arbeit sind für 20 MHz synthetisiert. Der Teiler entspricht also fünf. Damit die Software Zugriff auf den Oszillator hat, muss die von Xess dafür mitgelieferte svf-Datei *oscxsv.svf* auf das CPLD geladen werden. *Bei xsload* kann speziell für die Taktprogrammierung die Option *-DIV* mit dem Teiler angegeben werden. Der Aufruf, um das Board mit einem Takt von 20 MHZ zu programmieren, ist *xsload oscxsv.svf -DIV 5*.

Je nach dem, mit welcher Methode das FPGA personalisiert werden soll, muss ein anderes CPLD Design geladen werden. Um das FPGA mit einer bit-Datei

zu initialisieren, muss die Datei *downpar.svf* aus dem *cpld* Verzeichnis benutzt werden. Dieses CPLD-Design hat zu der von Xess mitgelieferten Version, in der Datei *dwnldpar.svf*, die serielle Schnittstelle zum FPGA durchgeführt. Nach erfolgreichem Laden des Designs auf das CPLD leuchten zwei Statusausgaben auf der LED-Strichanzeige. Die Daten, die an der parallelen Schnittstelle anliegen, werden auf einer der beiden 7-Segmentanzeige des Boards angezeigt. In der Abb. 3.1 ist dies die linke Anzeige. Nachgeprüft werden kann dies mit *xsport*. Ist das Design geladen kann mit einer bit-Datei das FPGA personalisiert werden. Nach dem Laden wird das Design sofort ausgeführt.

Soll das FPGA aus Daten von dem FlashRAM personalisiert werden, so müssen diese zuerst in das FlashRAM geschrieben werden. Mit dieser Konfigurationsmethode bleibt das Design nach einer Stromunterbrechung auf dem Board erhalten. Um Daten in das FlashRAM zu bekommen, muss das Xilinx Design *flashprg.svf* im Verzeichnis *xsvsoft* auf das CPLD geladen werden. Dies geschieht allerdings automatisch, wenn *xsload* eine exo-Datei zum Laden übergeben bekommt. Hierfür ist der richtig gesetzte Pfad in *XSTOOLS_BIN_DIR* wichtig. In der exo-Datei können die mit *promgen* erzeugten Daten des FPGA-Designs stehen (siehe 2.6). Sind die Designdaten der exo-Datei auf dem Flash gespeichert, muss auf das CPLD die Datei *flash_to_fpga.svf* aus dem Verzeichnis *cpld* geladen werden. Es enthält die zum Design *flashcfg.svf* von Xilinx geänderte serielle Verbindung und lädt nach Stromzufuhr das Design aus dem Flash in das FPGA. Für die serielle Verbindung wurden die letzten vier freien Leitungen vom CPLD zum FPGA verwendet, da alle anderen Leitungen für den Bootvorgang der CPU aus dem Flash benötigt werden.

Nach dem Laden des Designs auf das FPGA beginnt die CPU zu starten. Ist sie mit integriertem BPROM synthetisiert, führt sie den 1 kByte großen Bootloader darin aus. Dieser initialisiert die beiden Speicherkonfigurationsregister und gibt eine Bootmeldung auf der seriellen Schnittstelle aus. Um diese auf dem Rechner auszugeben muss ein Nullmodemkabel vom XSV-800 Board mit dem Linux PC verbunden werden. Mit einem Terminalprogramm wie *seyon* oder *minicom* werden diese auf dem Bildschirm dargestellt. Die Standardeinstellungen für die serielle Schnittstelle sind 38400 Baud mit acht Datenbits, keine Prüfsumme und ein Stopbit (8N1). Nach der Bootmeldung wartet der Bootloader auf die Übertragung eines auszuführenden Programms im S-Record-Format (siehe 2.9). Das Programm kann mit *cat <Programm.srec> /dev/ttyS0* übertragen werden. Es kann ebenfalls mit einem ASCII-upload aus den Terminalprogrammen geschehen. Anschließend wird das Programm auf LEON ausgeführt. Nach jedem Reset muss das Programm erneut übertragen werden.

Um das System aus dem FlashRAM zu booten, müssen die ROM-Daten in das obere MB des Flashs geschrieben werden. Ein entsprechendes ROM wird mit *mkprom* erzeugt. Für das RAM sind dabei keine (0) Waitstates nötig, da die

Zugriffszeit des RAMs (19 ns) kleiner als ein Takt (50 ns) ist. Da das Flash eine wesentlich größere Zugriffszeit (85 ns) hat, müssen bei diesem 2 Waitstates verwendet werden. Das erzeugte ROM kann anschließend mit *sparc-rtems-objcopy* in das S-Record Format gewandelt werden, welches von den exo-Dateien verwendet wird. Die Zieladresse muss dabei auf 0x100000 geändert werden, damit die Daten in das oberer MB des Flashs geschrieben werden. Zusätzlich muss die Initialisierungs- und Startanweisung aus dem S-Record entfernt werden. Ein fertiges Makefile zur Erzeugung einer solchen exo-Datei steht in *ddm/software*. Da *mkprom* in der derzeitigen Version noch nicht alle Features der Speicherschnittstelle unterstützt, wird auch der ROM-Code von dem Makefile angepasst. In dem ROM-Code muss die ROM-Breite auf 8 und Read-Modify-Write als aktiv in den Speicherkonfigurationsregistern von LEON gesetzt werden.

Kapitel 4

Diktiergerät als SoC

Der Entwurf eines SoCs mit der Leon-Plattform wird in diesem Kapitel anhand des Beispiels digitales Diktiergerät beschrieben. Das System hat als Aufgabe Audiodaten aufzunehmen und zu speichern, um sie zu einem späteren Zeitpunkt wieder auszugeben. Der Benutzer kann dabei wählen, ob er eine neue Aufnahme starten, eine alte Aufnahme abspielen oder diese löschen will.

Ein Prototyp des Geräts wurde auf das XSV-800 Board abgebildet. Es benutzt den auf dem Board befindlichen Audiobaustein (siehe Abschnitt 3.1). Dieser Baustein vom Typ AKM AK4520A [ASA97] kann Stereo-Audiosignale A/D und D/A wandeln. Die Audiodaten werden als serieller binärer Bitstrom zum FPGA transportiert. Zur Steuerung des Diktiergeräts werden die Mikrotaster des Boards benutzt, wodurch das System standalone betrieben werden kann. Als Statusausgabe dienen die beiden 7-Segmentanzeigen. Auf ihnen wird die Anzahl der vorhandenen Aufnahmen, als auch bei Aufnahme und Wiedergabe der aktuelle Index der Aufnahme, wiedergegeben.

4.1 Entwicklungszyklus

Die Abfolge der Entwicklung eines Cores für das System ist in Abbildung 4.1 dargestellt und wird folgend anhand des Beispiels Diktiergerät diskutiert. Die Systemfunktionalität wird als erstes in Hardware und der Software aufgeteilt. Die Hardware übernimmt die Ansteuerung der Peripherie und zeitkritische Aufgaben, sollte aber möglichst vielseitig nutzbar ausgelegt sein. Die Software implementiert auf der Hardware die eigentliche Funktion des Systems, in diesem Fall das Diktiergerät. Die Hardware- und Softwareentwicklung können nach der Spezifikation parallel erfolgen. In der Arbeit liefen diese aber nacheinander ab.

Als erstes sollte bei der Hardwareumsetzung ein Probedesign zur Ansteuerung der Peripherie entworfen werden. Für das Diktiergerät ist dies ein Testdesign zur

Audioansteuerung. Dieses ist in der Datei *soundtest.vhd* zu finden. Es kann direkt ohne das restliche System auf der Hardware getestet werden. Dadurch können schneller Fehler im Zeitverhalten der Signale aufgedeckt werden, wodurch der Entwicklungszyklus viel schneller abläuft. Nach erfolgreicher Ansteuerung wird die eigentliche Schaltung des Cores entworfen. Um die grundlegenden Funktionen der Hardware testen zu können, wird eine eigene Testumgebung dafür geschrieben. Damit wird die zeitaufwendige Simulation des gesamten Systems umgangen. Zu den Grundfunktionen zählt beim Diktierigerät die Generierung der Taktsignale für den Audiobaustein und das korrekte Schieben der Audiodaten. Die Schaltung enthielt bei diesem Test aber alle benötigten Komponenten und konnte jederzeit mit dem gesamten System übersetzt werden. Zur Simulation werden aus diesem Grund auch die Dateien *target.vhd*, *config.vd*, *device.vhd*, *sparcv8.vhd*, *amba.vhd*, *iface.vhd*, *ambacomp.vhd* aus der LEON-Plattform von der Schaltung benötigt. In dem Verzeichnis *ddm/design* sind alle Dateien mit Ausnahme des Hülldesigns, die an der Plattform geändert werden mussten, zu finden. Nachdem die grundlegende Funktionalität des Cores mit der eigenen Testbench geprüft ist, wird die Schaltung mit der gesamten Plattform getestet. Mit der Software wird die eigentliche Funktion des Systems als Anwendung implementiert. Neben der Anwendung sollte aber auch eine Software basierte Testbench geschrieben werden. Diese testet die Grundfunktionen der Hardware und das Zusammenspiel des Cores mit der Plattform. Mit der Systemsimulation können auf diese Weise Fehler in der Kommunikation zwischen System und integriertem Core untersucht werden. Dies betrifft vor allem den AHB Master, da dessen Funktionsfähigkeit bei der Testsimulation des einzelnen Cores nicht hervorgeht. Nachdem das System in der Simulation befriedigend läuft, wird das Design synthetisiert und auf der Hardware mit der Softwaretestbench getestet. Zuletzt wird die eigentliche Anwendung auf der Hardware in Betrieb genommen. Dabei werden vorallem Fehler der Software auftreten, da die Software auf Simulatorbasis nur unzureichend getestet werden kann. Es können aber auch noch Hardwarefehler auftreten, die erst auf dem realen System zur Geltung kommen.

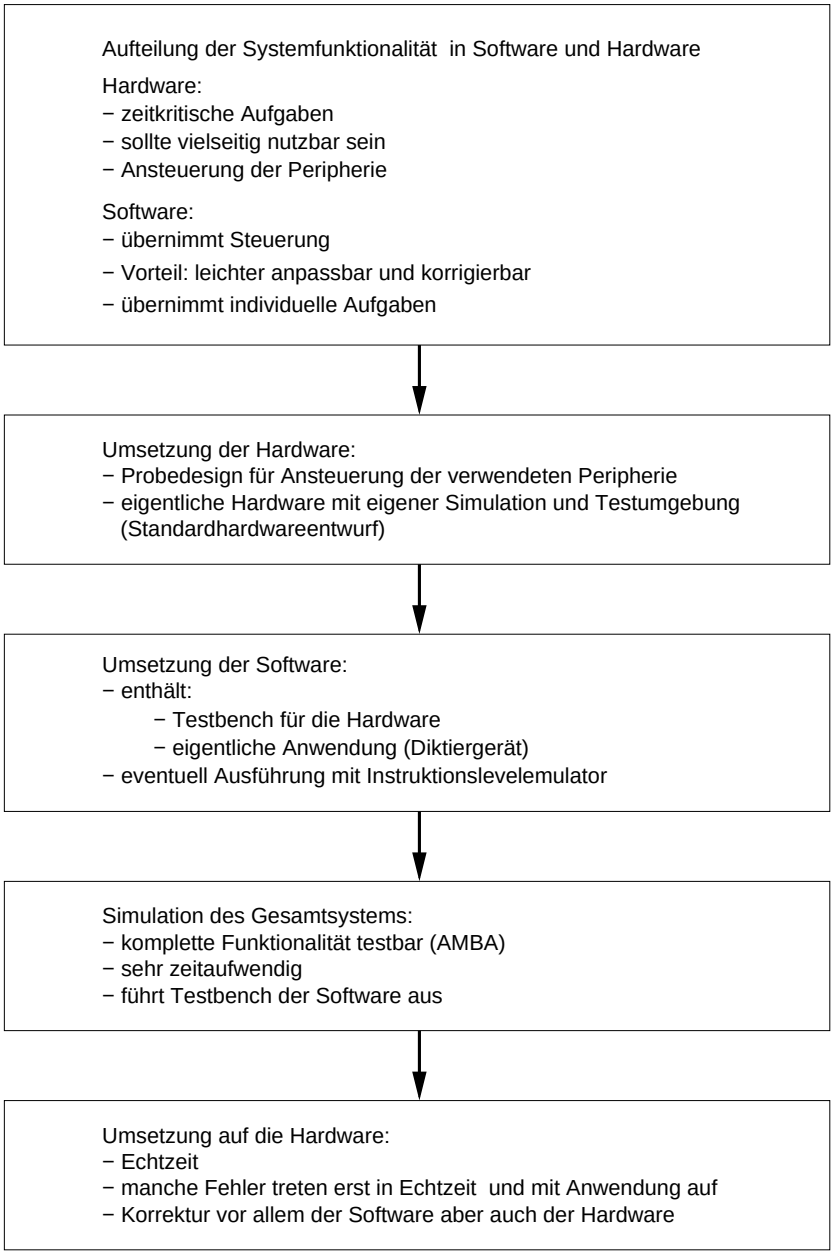
Aufteilung der Systemfunktionalität in Software und Hardware
Hardware:
− zeitkritische Aufgaben
− sollte vielseitig nutzbar sein
− Ansteuerung der Peripherie
Software:
− übernimmt Steuerung
− Vorteil: leichter anpassbar und korrigierbar
− übernimmt individuelle Aufgaben
Umsetzung der Hardware:
− Probedesign für Ansteuerung der verwendeten Peripherie
− eigentliche Hardware mit eigener Simulation und Testumgebung (Standardhardwareentwurf)
Umsetzung der Software:
− enthält:
− Testbench für die Hardware
− eigentliche Anwendung (Diktiergerät)
− eventuell Ausführung mit Instruktionslevelemulator
Simulation des Gesamtsystems:
− komplette Funktionalität testbar (AMBA)
− sehr zeitaufwendig
− führt Testbench der Software aus
Umsetzung auf die Hardware:
− Echtzeit
− manche Fehler treten erst in Echtzeit und mit Anwendung auf
− Korrektur vor allem der Software aber auch der Hardware

Abbildung 4.1: Entwicklungszyklus einer Anwendung auf der Plattform

4.2 Funktionsbeschreibung

In dem Core des Diktiergeräts übernimmt die Hardware die Aufgabe der Audio Ein- und Ausgabe, das Auslesen der Mikrotaster und die Ansteuerung der beiden 7-Segmentanzeigen. Der Audiobereich teilt sich in die Ansteuerung des AKM AK4520A und dem Schreiben und Lesen der Audiodaten in oder aus dem Speicher auf. Zur Speicheransteuerung wurde ein AHB Master integriert. Die Audioansteuerung ist wiederum in einen Timer und den Datenshifter für den seriellen Audiostrom trennbar. Die Hardware wird über Konfigurationsregister, die über den APB gelesen oder geschrieben werden, gesteuert. Eine Übersicht über den Digital Dictation Machine (DDM) Core ist in Abb. 4.2 zu sehen.

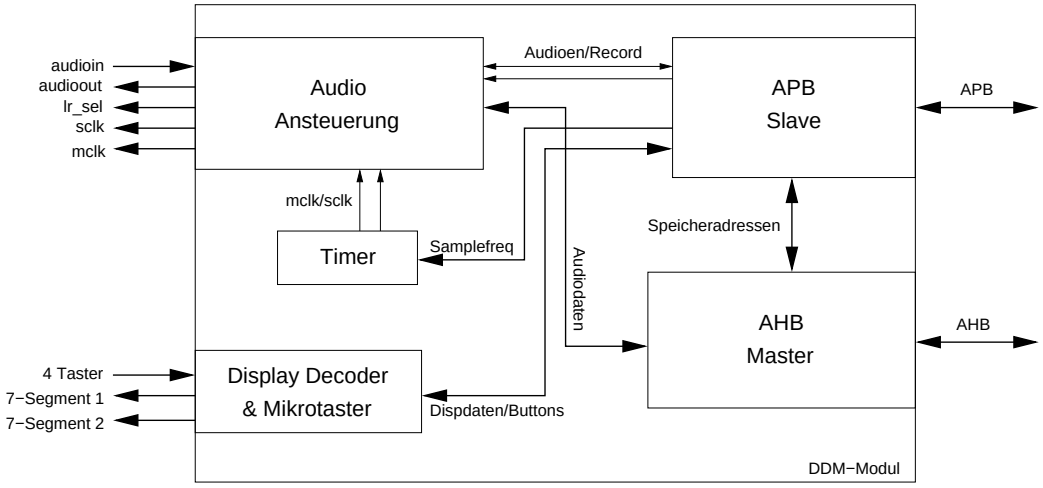

Abbildung 4.2: Übersicht über den DDM-Core

Die Software übernimmt die Steuerung der Hardware und die Verwaltung des vorhandenen Speichers. Sie kommuniziert dabei mit der dazugefügten Hardware über die Speicher abgebildeten Register. Über die Register liest sie den Zustand der Mikrotaster aus und wird dadurch vom Benutzer gesteuert. Die Software steuert die Audioaktionen und wohin die Audiodaten geschrieben oder woher diese gelesen werden. Sie gibt an, was auf den 7-Segmentanzeigen des Boards ausgegeben wird. Kurz gesagt ist die Hardware eine Ausführungsinstanz, die auch für andere Aufgaben benutzt werden kann, und die Software implementiert darauf das Diktiergerät.

4.3 Schaltungsaufbau

Die Schaltung wurde wie die LEON-Plattform in VHDL implementiert. Sie ist in einer einzelnen Datei mit Namen *ddm.vhd* im Verzeichnis *ddm/design* untergebracht. Der Entwurf teilt sich in drei Prozesse auf. Dem Timer-Prozess *timerpr*, der die Taktsignale zur Audioansteuerung erzeugt, und dem *ddmtop*, in dem jede andere Logik enthalten ist. Der Prozess *regs* ist für die Ansteuerung des Systemtakts an den Registern verantwortlich. Um die Funktionsweise der Schaltung zu erklären, ist sie in die Bereiche der Abb. 4.2 aufgeteilt. In der realen Schaltung sind diese jedoch nicht so klar getrennt und die Trennung dient lediglich der Übersichtlichkeit. Im Folgenden werden zuerst die erforderlichen Signale zur Ansteuerung des Audiobausteins AKM AK4520A erklärt.

4.3.1 Die Signale des AKM AK4520A

Der AKM AK4520A [ASA97] kann ein Stereo-Audiosignal mit einer Samplefrequenz von max. 48 kHz und einer Auflösung von 20 Bit simultan aufnehmen und abspielen. Die digitalen Audiodaten werden dazu seriell in den Baustein hinein oder aus ihm heraus befördert. Zur Synchronisation benötigt er einen Mastertakt *MCLK*, ein Shifttakt *SCLK* und einen Takt, der zwischen rechtem und linken Kanal umschaltet *LRCK*. Von diesen Takten hängt die benutzte Samplefrequenz ab. Wenn der Takt *LRCK* nicht durchgehend anliegt wird ein Reset auf dem Baustein ausgeführt. Von der Schaltung werden deshalb alle Takte durchgehend ausgegeben, damit das System immer zu Audioaktionen bereit steht. Der Audiobaustein kann in verschiedenen Modi betrieben werden, die durch die Eingangssignalen *DIF0*, *DIF1* und *CMODE* ausgewählt werden. *DIF0* und *DIF1* sind auf dem XSV-800 Board fest mit “10” angesteuert. Dies entspricht Modus zwei, und bedeutet, dass die Audiodaten mit 20 Bit (hochwertigstem Bit zuerst) bei der Ein- und Ausgabe geschoben werden. *CMODE* steht fest auf ’1’ und heißt, dass der *MCLK* das 256-fache der Samplefrequenz betragen muss. Die Audiodaten werden über die Leitungen *SDTI* und *SDTO* in und aus dem Baustein geschoben. Das genaue Zeitverhalten der Signale ist in der Abbildung 4.3 aus der AK4520A Spezifikation [ASA97] zu sehen. Das Verhältnis vom SCLK zur Samplefrequenz ist nicht fest vorgegeben und muss lediglich über das 40 fache zur Samplefrequenz betragen. Um den Takt gut mit dem MCLK synchronisieren zu können, wurde der SCLK ein Viertel so schnell wie dieser gewählt. Somit beträgt der SCLK das 64 fache der Samplefrequenz. Da nur 20 Bit an Audiodaten pro Kanal anfallen, werden jeweils 12 Bit ’don’t cares’ nach den eigentlichen Daten geschoben. Dadurch erzeugt die Schaltung die 32 Bit pro Kanal ($2 * 32 = 64$).

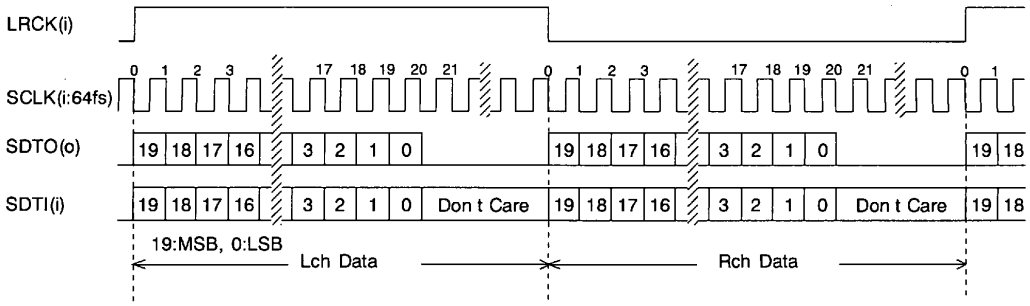

Abbildung 4.3: Das Zeitverhalten der Taktsignale des AK4520A

4.3.2 APB Slave

Über den APB Slave kommuniziert die CPU mit der Schaltung. Die Konfigurationsregister des Cores sind in den Speicherbereich des Systems gelegt und können von der Software gelesen und geschrieben werden. Über diese wird der DDM-Core gesteuert. In der Zeichnung 4.4 ist zu sehen, dass nicht alle Register gelesen und geschrieben werden können. Die Register werden in den weiteren Modulen des Cores verwendet und sind in den folgenden Abschnitten beschrieben. Der APB arbeitet nach dem in Abschnitt 2.10.1 beschriebenen Protokoll.

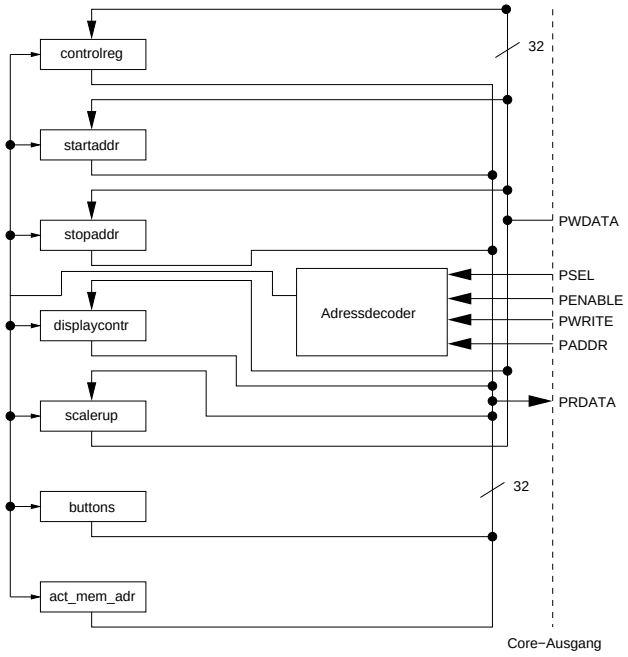

Abbildung 4.4: Schaltung des APB Slave

4.3.3 Mikrotaster und Hexdisplay

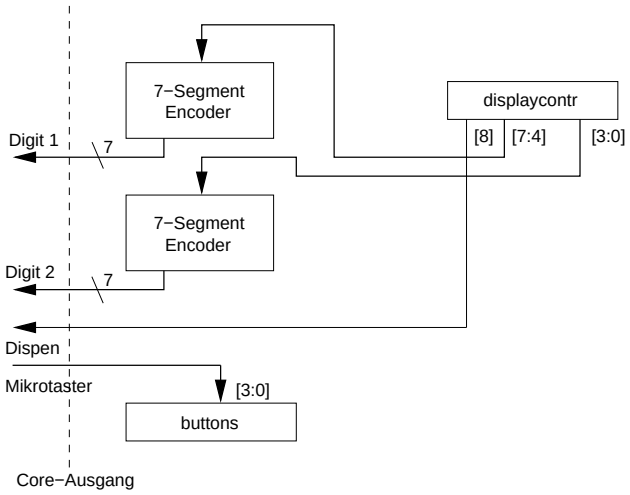

Abbildung 4.5: Schaltung Mikrotaster und Hexdisplay

Das Schaltbild zur Ansteuerung der Mikrotaster und der 7-Segmentanzeigen ist in Abb. 4.5 zu sehen. Der Zustand der vier Taster wird auf das vier Bit große Register *buttons* abgebildet, das über den APB von der Software ausgelesen werden kann. Von den vier möglichen Tastern des Boards, um das Diktiergerät zu steuern, wird der Taster0 als Resetknopf für das System verwendet. Es sind deswegen nur Bit 0 bis 2 mit Tasten verbunden. Bit 3 wird vom Hülldesign konstant auf ’0’ gesetzt. Auf den beiden LED-Anzeigen wird der Inhalt aus den unteren 8 Bit des Registers *displaycontr* über zwei 7-Segmentencoder ausgegeben. Bit 9 des Registers wird von dem Hülldesign zum Ein- und Ausschalten der Anzeigen benötigt und wird später genauer beschrieben.

4.3.4 Timer

Der Timer ist für die Erzeugung der Taktsignale des Audiobausteins zuständig. Mit ihm wird der Mastertakt *mclk* und der Schiebetakt *sclk* erzeugt. Um unterschiedliche Samplefrequenzen mit dem Design benutzen zu können, wurde ein Zähler verwendet. Dieser zählt den Wert aus *scalerup* herunter und löst beim Null Durchlauf des Zählers einen Tick aus. Bei jedem Tick des Zählers wird der Wert im Register *mstclk* negiert. Der Ausgang des Registers *mstclk* ist der Mastertakt *mclk*. Ist der Wert in scalerup auf 0 gesetzt, wird die Systemfrequenz halbiert. Die Frequenz des Mastertakt kann über die Formel $mclk = \frac{Systemfrequenz}{2+2*scalerup}$ bestimmt werden. Um eine Taktansteuerung des Audiobausteins im Bereich der Samplefrequenz, also im kHz Bereich, zu erzeugen wurde der Timer mit einem

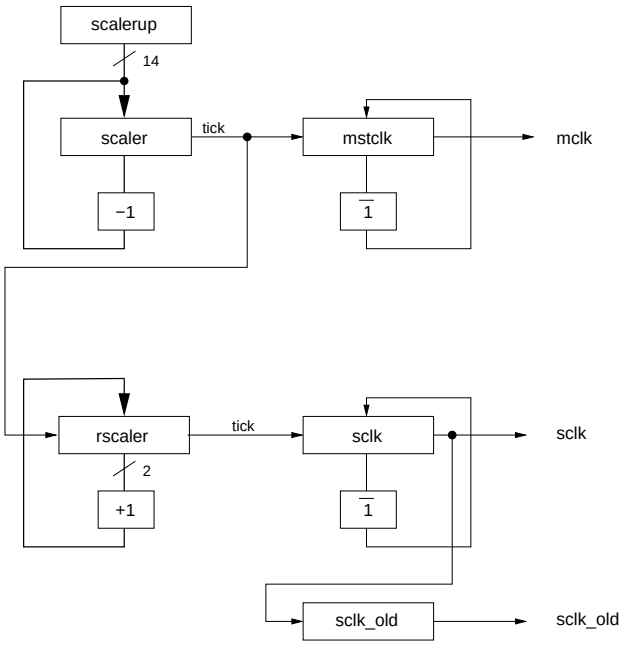

Abbildung 4.6: Schaltung des Timers

14 Bit breiten Zähler ausgelegt. Da der Takt des *mclk* des Audiobausteins aber den 256 fachen Takt der Samplefrequenz benötigt, liegt dieser im MHz Bereich. Der AK4520A ist für Frequenzen von 4096 bis 13824 kHz für den *mclk* spezifiziert. Das bedeutet bei einem Systemtakt von 20 MHz, dass nur die Werte 0 (10 MHz) und 1 (5 MHz) für das Register *scalerup* sinnvoll sind. Es ist daher für diese Aufgabe etwas großzügig ausgelegt. Während der Tests wurde aber auch ein Wert von 2 erfolgreich eingesetzt.

Aus den Ticks des Zählers wird auch der Schiebetakt generiert. Dazu zählt ein 2 Bit breiter Zähler *rscaler* die Ticks vom Zähler *scaler*. Steht der *rscaler* auf 0 wird ein Tick, mit dem der Inhalt des Flipflop *sclk* negiert wird, ausgelöst. Der *sclk* hat dadurch genau die benötigte ein Viertel so schnelle Frequenz des Mastertakts. Damit die Logik im Prozess *ddmtop* auf die steigende Flanke des *sclk* reagieren kann, wird es durch ein weiteres FlipFlop gesteuert. An diesem *sclk_old* liegt der Wert aus *sclk* immer einen Systemtakt später an.

4.3.5 Audio Ansteuerung

In Abb. 4.7 ist die vereinfachte Schaltung zur Audioansteuerung zu sehen. Die Audiodaten werden in das oder aus dem 20 Bit breiten Schieberegister geschoben. Über die Steuerlogik wird entschieden, wann Daten geschoben werden und ob diese in das Schieberegister hinein oder aus ihm heraus geschoben werden.

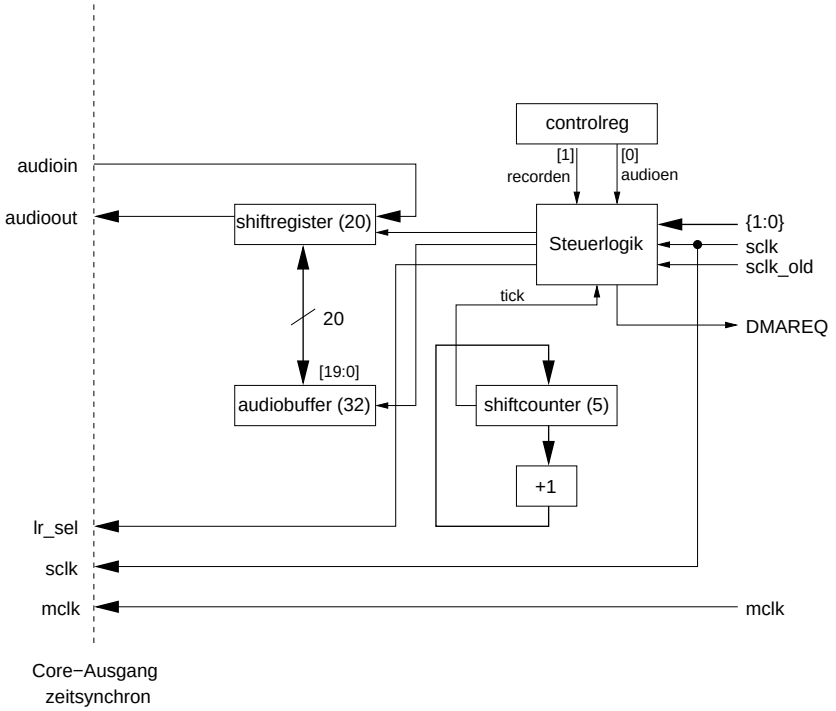

Abbildung 4.7: Schaltung der Audio Ansteuerung

Geschoben wird jeweils an der steigenden Flanke des Schiebetakts, was über die Leitungen *sclk* und *sclk_old* ausgewertet wird. Die Anzahl der Schiebevorgänge wird in dem 5 Bit breiten Register *shiftcounter* gezählt. Dieses zählt genau die 32 Schiebevorgänge eines Samples pro Kanal. Beim Durchlauf des Zählers wird der Kanal über *lr_sel* gewechselt. Daten werden nur aus oder in das Schieberegister geschoben, wenn die entsprechenden Bits *audioen* und *recorden* im Speicher abgebildeten Register *controlreg* gesetzt sind. Mit diesen kann eine Wiedergabe oder Aufnahme gestartet werden. Nach 20 Schiebevorgängen eines Kanals wird ein Schiebestop gesetzt. In diesem Fall wird konstant eine ’0’ am Audioausgang angelegt. Erst beim Wechsel von *lr_sel* auf den linken Wiedergabe Kanal wird der Schiebestop wieder gelöscht. Der Core arbeitet nur mit einem Audiokanal, ist also Mono. Bei der Wiedergabe auf dem linken Kanal liegen die Daten zur Aufnahme aus dem rechten Kanal; AK4520A arbeitet simultan; an. Der Core hat dadurch das Verhalten auf dem rechten Kanal aufzunehmen und auf dem linken abzuspielen. Dieses Verhalten wurde für eine leichtere Implementierung in Kauf genommen. Nach dem Schieben der eigentlichen Audiodaten werden die Daten bei einer Aufnahme in das 32 Bit Register *audiobuffer* kopiert. Bei der Wiedergabe werden diese von dort in das Schieberegister gelesen. Anschließend wird ein DMA-Transfer mit DMAREQ ausgelöst. Diese Anfrage wird vom AHB-Master

ausgeführt.

Alle Signale zum Audiobaustein werden zeitgleich auf den Ausgängen angelegt. Dazu werden die Signale zum Teil noch einmal zwischengespeichert.

4.3.6 AHB Master

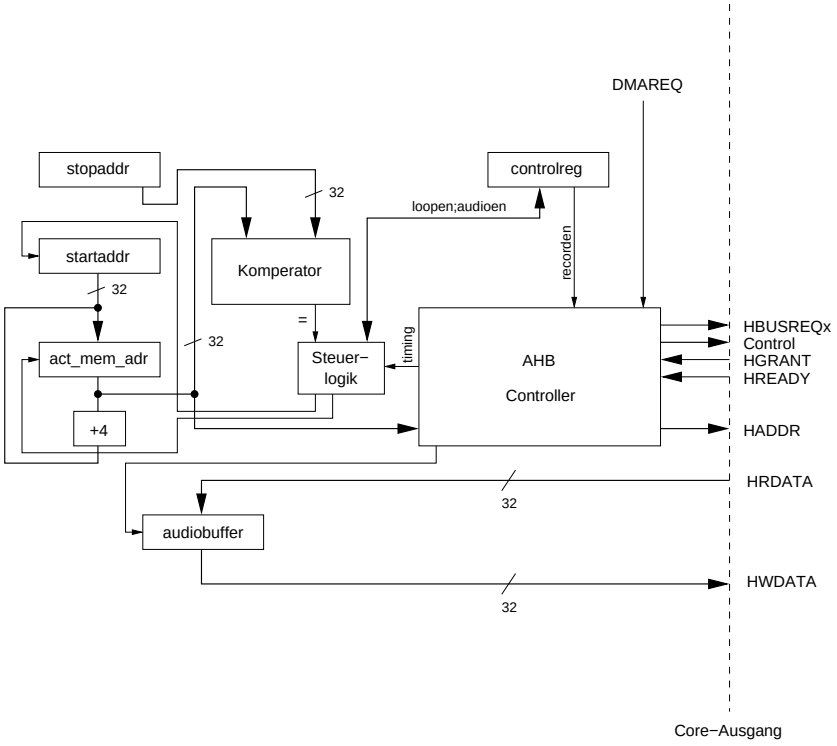

Abbildung 4.8: Schaltung des AHB-Master

Der AMBA-Master ist für das Schreiben und Lesen der Audiodaten in oder aus dem Speicher verantwortlich. Bei jeder DMA-Transferanfrage durch die Audioansteuerung fordert er die Kontrolle über den AMBA-Bus vom Arbiter an. Nachfolgend wird das Protokoll aus Abschnitt 2.10.2 benutzt. Als Speicheradresse wird der Wert aus dem Register *act_mem_adr* benutzt. Nach erfolgtem Datentransfer über den Bus wird die Adresse um 4 erhöht und mit dem Wert in dem Speicher abgebildeten Register *stopaddr* verglichen. Ist die Stopadresse erreicht, wird das *audioen* Bit in *controlreg* gelöscht und damit die Audioaktion beendet. Ist aber das *loopen* Bit in *controlreg* gesetzt, wird aus dem Speicher abgebildeten Register *startaddr,* das Register *act_mem_adr* neu geladen. Dies geschieht auch beim Aktivieren des Audiocores durch Setzen des Bit *audioen* in *controlreg*.

4.4 Integration in die LEON-Plattform

Zur Integration des fertigen DDM-Cores wird die LEON-Plattform folgendermaßen angepasst. Die Ein- und Ausgangsleitungen des Cores aus dem Gesamtsystem werden in einem Record zusammen gefasst und in der Datei *iface.vhd* eingetragen. Die Typen und Leitungsnamen für die Einbindung der AMBA-Komponenten sind in *amba.vhd* zu finden. Die neuen AMBA-Module des Busses werden in die Datei *ambacomp.vhd* eingetragen. Die Entity der neuen Schaltung wird in der Datei *mcore.vhd* an das System angeschlossen. In dieser Datei wird auch der Interruptcontroller mit dem Core verbunden. Alle Leitungen, die aus dem System herausführen, werden an die *leon* Entity weitergeben. Die *leon* Entity wird dazu in *leon.vhd* entsprechend verändert. Um es auf dem XSV-800 betreiben zu können, muss auch die Hülle um die LEON-Plattform verändert werden. Da das Design auch standalone betrieben wird und dazu das FlashRAM benutzt, gibt es eine spezielle Ansteuerung für die LEDs, das ROM und das RAM. Die Leitungen auf dem XSV-800 Board zum Flash und zu den LEDs der 7-Segementanzeigen sind dieselben. Am Anfang muss über diese der ROM Code geladen werden. Nachdem das Programm im RAM ausgeführt wird, können über die Leitungen die LED-Anzeigen angesteuert werden. Dazu gibt es in dem *ddm.vhd* Design das Ausgangssignal *dispen*. Über diese Leitung kann die Software zwischen ROM Betrieb und Anzeige Betrieb umschalten. In dem Hülldesign ist die Entity *selector2*; eine Weiterentwicklung von der Entity *selector*; für die Umschaltung zwischen ROM und Anzeige verantwortlich.

Damit das System korrekt übersetzt und synthetisiert wird, sind die Buildskripte entsprechend angepasst. Die AMBA-Konfiguration ist für den AHB Master und den APB Slave in der Datei *target.vhd* entsprechend geändert. Die AHB Master ist auf die Anzahl 2 erhöht worden. In der Liste der APB-Slaves wurde ein neuer Eintrag erstellt und der Adressbereich der Register eingetragen; 0x200 - 0x208.

4.5 Aufbau und Benutzung der Software

Die Software für das Diktiergerät ist ausschließlich in C geschrieben und besteht aus zwei Dateien. In der Programmdatei *ddm.c* befinden sich das eigentliche Programm, das in einige Funktionen aufgeteilt ist. Die Headerdatei *ddm.h* enthält die Definition der C-Struktur *ddmregs* (siehe Anhang D), die für den Zugriff auf die Speicher abgebildeten Register zuständig ist.

4.5.1 Zugriff auf die Register und ihre Funktion

Die Variablen der C-Struktur *ddmregs* spiegeln die Register in der Hardware wieder. Die Variablen sind alle vom Typ *unsigned int*, welcher wie der AMBA-Bus 32 Bit breit ist. Damit die einzelnen Variablen in der *structure* nicht vom Compiler in der Reihenfolge geändert werden können, muss die Anweisung *volatile* davor gestellt werden. Diese verhindert das Optimieren der nachfolgenden Variablen durch den Compiler. Durch das Schreiben des Wertes *REGSTART*, welcher die Anfangsadresse der Register im Speicher enthält, in einen Zeiger auf die *structure*, werden Zugriffe auf die Variablen dieser *structure* in Zugriffe auf die dazugehörigen Hardwareregister.

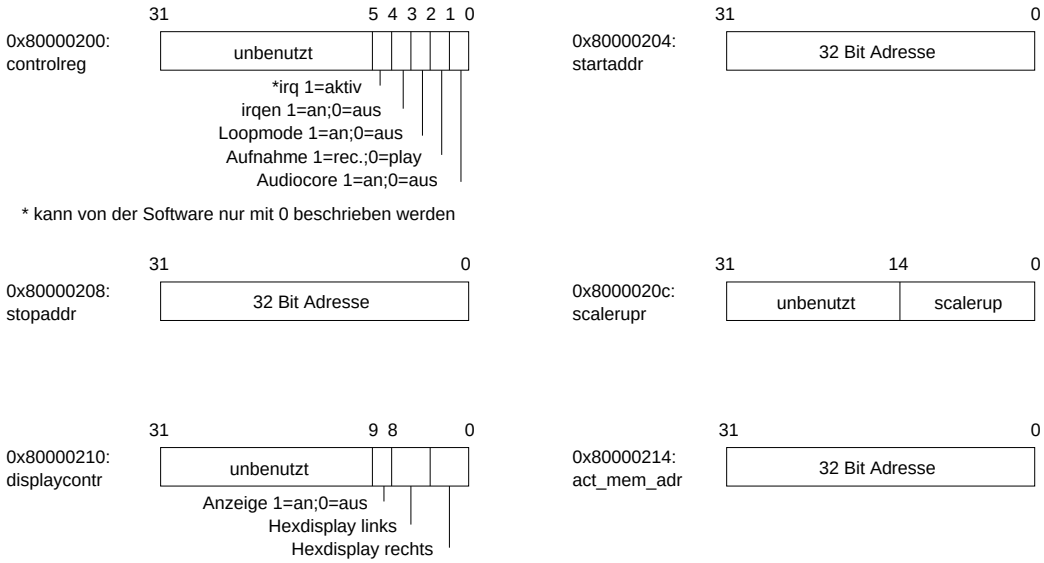

Abbildung 4.9: Belegung und Adressen der Speicher abgebildeten Register

In Abb. 4.9 kann die Belegung und Funktion der Register, die ab Adresse 0x800000200 im Speicher liegen, abgelesen werden.

4.5.2 Die Funktionen der Software

Die Software enthält die Testbench für die Hardware und die eigentliche Anwendung des Diktiergeräts. Zur Ansteuerung der Hardwarefunktionen sind die Zugriffe der Register in einzelne Funktionen verpackt worden. Diese und einige weiterer Hilfsfunktionen werden nachfolgend erklärt.

get_block() Wegen der fehlenden dynamischen Speicherverwaltung wurde eine eigene vereinfachte Speicherverwaltung geschrieben. Diese arbeitet

auf einem statischen Array. Zur Initialisierung muss am Anfang des Programms die Funktion *init()* aufgerufen werden. Mit *get_block()* kann ein 8 kByte großer Speicherblock des Arrays angefordert werden.

free_block() Sie gibt einen mit *get_block()* reservierten Speicherbereich wieder frei.

startrec() Sie startet eine Aufnahme auf der Audiohardware. Der Funktion muss die Anfangs- und Endadresse der Aufnahme und ein Wert zur Konfiguration des Loopmodes übergeben werden.

startplay() Sie startet die Wiedergabe einer Aufnahme auf der Hardware. Die Parameter sind die gleichen, wie bei der Aufnahme.

stop() Sie hält die Audiohardware sofort an.

hexddisp() Sie gibt den übergebenen Wert auf dem Hexdisplay der Hardware aus.

dispoff() Sie schaltet das Hexdisplay wieder aus. Zugriffe auf das ROM sind anschließend wieder möglich.

set_sample_freq() Mit dieser Funktion wird der Wert des Registers *scalerup* gesetzt. Der Wert wird dabei nicht umgerechnet. Der Name der Funktion ist deswegen etwas irreführend.

wait_for_audiofinish() Die Funktion wartet auf das Beenden einer Audioaktion und blockiert so lange die Ausführung des Programms.

get_buttons() Sie liest den Zustand der Mikrotaster aus der Hardware aus. Da es bei schnellem Programmfluss unter Umständen zu sehr schnellem Auslesen der Taster kommen kann, wird der letzte Zustand der Taster in einer statischen Variable gespeichert. Nur bei Änderung von ’0’ auf ’1’ eines Tasters wird die Aktivierung dieses Tasters von der Funktion weitergegeben.

trans() Die Funktion kopiert den Inhalt eines 8 kByte Speicherbereichs in einen anderen. Der Funktion werden dazu die Anfangsadressen der Speicher übergeben. Diese Funktion dient als Schnittstelle, falls die von der Hardware gelieferten Audiodaten im Speicher reduziert werden sollen.

trans_back() Dies ist die Umkehrfunktion von *trans()*. Sie kopiert den Speicherinhalt eins Puffers wieder zurück.

clear_buff() Diese Funktion löscht eine verkettete Liste von Speicherblöcken einer Aufnahme und gibt den Speicher durch Aufruf von *free_block()* wieder frei.

Um die Funktionsfähigkeit der Hardware testen zu können wurde eine Testbench in der Software implementiert. Diese steht in der Funktion *test()*. Sie greift in großen Teilen direkt, also ohne Benutzung der oben genannten Funktionen, auf die Register zu. Die einzelnen Zustände werden als Output auf der seriellen Schnittstelle ausgegeben. Als erstes testet sie den Zugriff auf die Register über den APB. Sie schreibt dazu einige Werte in die Register und liest diese anschließend wieder aus. Dann gibt sie die Zahlen von 0x00 bis 0xff in 0x11 Schritten auf dem Hexdisplay aus. Dies kann auch auf der realen Hardware, ohne Verbindung zu einem Computer, beobachtet werden. Zum Schluss überprüft sie die Funktionstüchtigkeit der Audioansteuerung und des AHB in dem sie eine Aufnahme startet. Die Daten werden in den Speicher geschrieben und nach Beenden der Aufnahme wieder ausgegeben. Kommt die Testbench am Ende an, gibt sie auf der Hexadezimalanzeige die Ziffern 0x0c aus und endet in einer Endlosschleife. Die Testbench sollte zur Benutzung mit der Simulationsumgebung oder der Hardware, entsprechend angepasst werden. Aus Zeitgründen sollten bei der Simulation alle Verzögerungsschleifen entfernt und der Puffer für die Aufnahme auf ein paar Sample begrenzt sein.

Die Anwendung, die das Diktiergerät implementiert, baut sich aus weiteren drei Funktionen auf. *ddm()* enthält die Hauptschleife. Diese Endlosschleife enthält die Steuerung der einzelnen Aktionsmöglichkeiten des Diktiergeräts. Am Anfang der Funktion wird die Plattform initialisiert. Der Funktion steht zur Speicherung der Aufnahmen ein Array mit fester Anzahl Einträge zur Verfügung. In diesem werden die Zeiger auf die einzelnen Aufnahmen gespeichert. In der Hauptschleife werden die Tastenzustände ausgewertet und die entsprechende Aktion gestartet.

Eine Aufnahme des Diktiergeräts besteht aus einer verketteten Liste von 8 kByte großen Speicherblöcken. In dem letzten 4 Byte eines Blocks wird der Zeiger auf den nächsten Block gespeichert. Mit *ddm_record()* wird eine solche Liste als Aufnahme angelegt. Dazu benutzt die Funktion zwei feste Puffer zur Aufnahme. Diese werden abwechselnd mit Daten gefüllt und jeweils mit *trans()* in einen dynamisch allokierten Speicherblock übertragen. Sind keine dynamischen Blöcke mehr verfügbar oder greift der Benutzer ein, wird die Aufnahme gestoppt. Um eine Aufnahme oder Wiedergabe ohne Sprünge mit den Blöcken zu realisieren wird der Loopmode der Hardware benutzt. Dieser springt bei erreichen der Endadresse wieder auf die Startadresse zurück. Durch Austauschen der Start- und Stopadresse während des Betriebs, kann so eine kontinuierliche Aufnahme erreicht werden. Um die richtigen Zeitpunkte zur Aktualisierung der Start- und

Stopadresse abzupassen, wird aus dem Register *act_mem_adr* die aktuelle Speicheradresse ausgelesen. Am Ende der Aufnahme wird von der Funktion ein Zeiger auf die Aufnahme zurückgegeben, welcher in dem Array der *ddm()* Funktion gespeichert wird. Die Wiedergabe funktioniert gleich der Aufnahme, nur dass die einzelnen Aufnahmeblöcke mit *trans_back()* zurück übertragen und anschließend abgespielt werden. Dazu wird die Funktion *ddm_play()* mit Übergabe des Zeigers auf die Aufnahme aufgerufen.

In der Funktion *main()* wird entweder die Testbench mit Aufruf von *test()* oder die Anwendung mit Aufruf von *ddm()* gestartet.

4.5.3 Benutzung der Software

Das Programm kann mit dem Makefile in dem Verzeichnis *ddm/software* übersetzt werden. Dabei wird auch die für den Betrieb mit dem FlashRAM benötigte exo-Datei erzeugt. Die Diktiergerätanwendung gibt Kontrollausgaben auf der seriellen Schnittstelle aus. Gesteuert wird sie über die Taster des Boards. Die Bedienung kann Abb. 4.10 entnommen werden. Der aktuelle Index der Aufnahme oder Wiedergabe wird in dem Hexdisplay angezeigt.

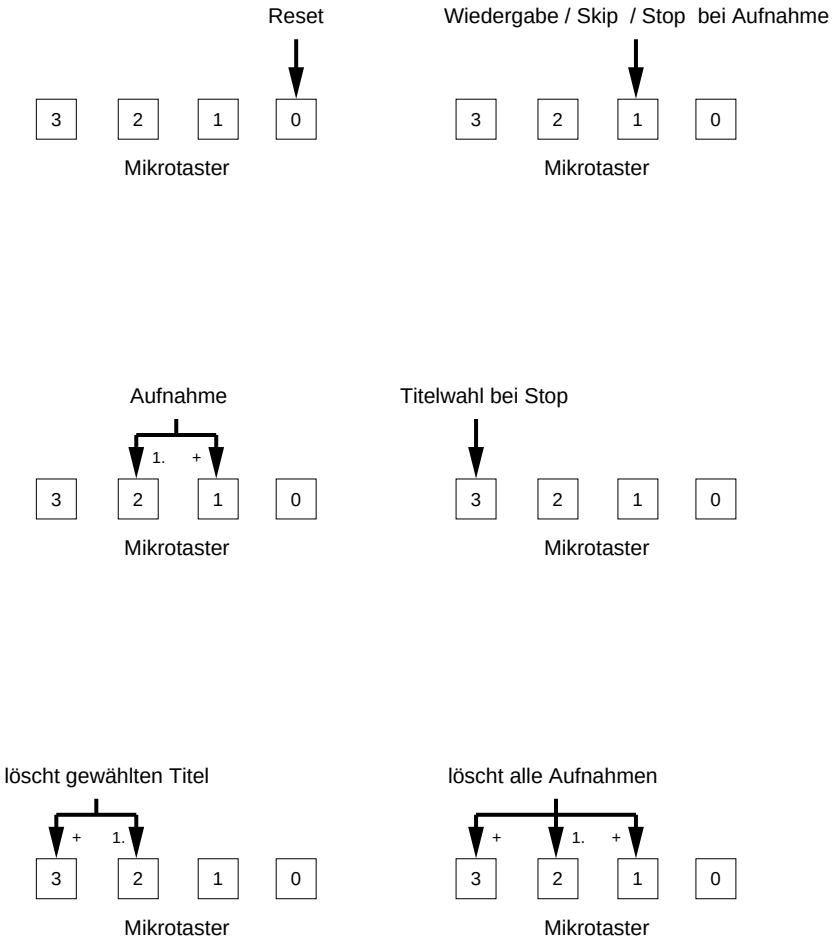

Abbildung 4.10: Bedienung des Diktiergeräts

Kapitel 5

Zusammenfassung und Ausblick

In der Arbeit wurde eine System-on-a-Chip Entwicklungsplattform für die Abteilung Rechnerarchitektur zusammengestellt. LEON wurde aus den auf dem Markt erhältlichen eingebetteten Systeme als Plattform ausgesucht. Sie wurde an die Designwerkzeuge der Abteilung angepasst. Die LEON-Plattform wurde auf dem ausgewählten XSV-800 Hardware-Prototypenboard in Betrieb genommen. Um die Entwicklung eines SoC mit der Plattform zu demonstrieren, wurde ein digitales Diktiergerät darauf implementiert. Dazu wurde ein eigener Core entwickelt und über den AMBA-Bus mit dem System verbunden. Mit der dafür geschriebenen Software wurde das Diktiergerät auf dem Board erfolgreich umgesetzt. Es kann ohne einen externen Rechner als eigenständiges System betrieben werden.

Die Arbeit zeigt neben der Umsetzung auch einen interessanten Einblick in den neuen Zweig der freien Hardwareentwicklung. LEON, als die erste Open Source CPU, steht erst am Anfang ihrer Möglichkeiten. So kann durch hohe Verbreitung und der billigen Umsetzung auf FPGA basierter Hardware, eine sehr schnelle Entwicklung erreicht werden. Leistungsstarke Umsetzungen für Dividierer und Multiplizierer sind schon in naher Zukunft zu erwarten. Änderungen der Cachearchitektur als teil- oder vollassoziativen Ausführung sind angekündigt worden. Varianten mit niedriger Leistungsaufnahme werden diskutiert. Die Implementierung einer FPU und MMU werden auch in ferner Zukunft kommen. Dank der SPARC Kompatibilität ist es leicht, Software auf das System umzusetzen. So ist nach Berichten neben dem mitgelieferten RTEMS Betriebssystem, μCLinux zum größten Teil auf der Plattform einsatzbereit. Neben den freien Entwicklern, haben aber auch kommerzielle Firmen Produkte mit LEON angekündigt. LEON steht damit eine große Zukunft offen. Ob LEON jedoch eine Konkurrenz zu kommerziellen Systemen wie ARM, MIPS, usw. sein kann, wird sich erst zeigen müssen. Um im Bereich der eingebetteten Systeme konkurrieren zu können, muss LEON für ASIC Prozesse synthetisiert werden. Für die Technologie Atmel ATC35 ist das Design vorbereitet. Durch die Offenheit des

Systems neue Technologien zu integrieren kann es aber auch leicht an neue angepasst werden. Firmen wie Metaflow [met01] haben schon Produkte mit LEON angekündigt. Doch ist LEON den aktuellen, von Firmen entwickelten Systemen, noch um einige Schritte hinterher.

Anhang A

CVS

CVS (Concurrent Versions System) [Fog99] ist ein Verwaltungssystem für Quelltexte. Es ermöglicht verschiedene Versionen der Quelltextdateien zu archivieren und verwalten. Dazu gibt es einen CVS-Server auf dem die Daten gelagert werden. Der Benutzer, meist ein Softwareentwickler, kann von diesem Dateien beziehen (check out) oder solche auf ihm archivieren (check in). Die Versionskontrolle wird dabei vollständig von dem Server übernommen. Dies erleichtert die Arbeit eines, aber vor allem die mehrerer Entwickler, erheblich. Die Programmcodedateien eines zusammengehörigen Systems werden dabei immer in dem dazugehörigen Projekt gespeichert, welches am Anfang der Entwicklung erstellt werden muss.

A.1 Umgang mit CVS

Um mit CVS arbeiten zu können, muss das CVS-Programm wissen woher und wie die Daten bezogen werden. Dies wird entweder direkt beim Ausführen des Befehls *cvs* mit der Option *-d* oder durch Setzen der Umgebungsvariable *CVSROOT* angegeben. Die Syntax ist dabei

:<logintype>:<username>@<host>:<path>

wobei

<logintype> = den Typ des Login angibt

<username> = der Username des Benutzers auf dem Host

<host> = der Rechnername auf dem der CVS-Server läuft

<path> = der Pfad zu den Repository auf dem Server

ist.

Der *logintype* gibt dabei die Art der Verbindung vom Benutzerrechner zum Host an. Wird *pserver* (password-authenticated server) angegeben, wird ein *login* ausgeführt. Das angegebene Passwort wird dabei im Home-Verzeichnis in der Datei *.cvspass* gespeichert und muss beim nächsten Ausführen von cvs nicht mehr angegeben werden. Ein anderer Typ ist *ext* (external connection program). Dabei wird das in der Umgebungsvariable *CVS_RSH* gesetzte Loginprogramm benutzt. In der Umgebung der Abteilung Rechnerarchitektur ist dies die *ssh* (secure shell).

Kommandos von CVS werden mit *cvs <command>* ausgeführt. Eine Übersicht der Befehle in CVS kann mit der Eingabe *cvs commands* ausgegeben werden. Die wichtigsten Befehle zur Arbeit mit CVS sind:

cvs import <project name> <vendortag> <releasetag>

> Beginnt ein neues Projekt mit Namen *project name*. Der *vendortag* gibt dabei den Ersteller des Quelltextes an. Die Version wird mit dem *releasetag* angegeben. Mit der Option *-m* kann ein Kommentar in Anführungszeichen angegeben werden, welcher sonst mit dem Standardeditor eingegeben werden soll. In dem neu eröffneten Projekt werden alle Dateien und Verzeichnisse des aktuellen Pfades gespeichert.

cvs checkout <project name>

> Lädt die zu einem Projekt gehörenden Dateien und Verzeichnisse, in der aktuellsten Version, vom Server in den aktuellen Pfad.

cvs diff [file name]

> Gibt die Unterschiede zwischen der optional angegebenen Datei(en) und der aktuellen Version dieser Datei(en) auf dem CVS-Server an. Wird keine Datei angegeben, werden alle in dem aktuellen und in den Verzeichnis darunter befindlichen Dateien untersucht. Statt einer Datei kann auch ein Verzeichnis oder auch mehrere Dateien angegeben werden.

cvs update [file name]

Dabei werden mit Hilfe eines *diff* die Unterschiede zwischen der lokalen und der auf dem Server befindlichen Datei ermittelt. Alle Änderungen, auf Serverseite seit dem letzten Auschecken der Datei auf den lokalen Rechner, werden dann automatisch in die lokale Version übernommen. Entstehen dabei Konflikte, die das System nicht lösen kann, wird der Benutzer darauf aufmerksam gemacht und aufgefordert, dies zu tun.

cvs add <file name>

Hiermit wird dem CVS-Server eine neue Datei (Verzeichnis) bekannt gemacht. Dies ist nötig, damit CVS beim Einchecken nicht alle in einem Verzeichnis befindlichen Dateien einchecked, sondern nur die vom Entwickler als zum Projekt dazugehörend angegebenen. So wird verhindert, dass sich in den Verzeichnissen befindliche Binaries, Objektfiles und sonstige Dateien auf dem Server landen.

cvs commit [file name]

Hiermit wird eine lokale Datei auf den Server eingechecked. Dazu wird die Datei mit einem *diff* auf Veränderungen zum Server untersucht und gegebenenfalls die neue Version auf den Server gelegt. Ist zwischenzeitlich auch die Datei auf dem Server geändert worden, so wird die lokale Version vorher mit einem *update* entsprechend angepasst. Bei diesem Kommando muss auch ein Kommentar angegeben werden, welcher entweder mit der Option *-m* oder mit dem Standardeditor eingegeben wird.

A.2 Die Projekte der Entwicklungsumgebung

Als CVS-Server dient in der Abteilung Rechnerarchitektur der Rechner *rai16*. Als Useridentifikation wird der eigene Loginname benutzt, in diesem Fall *bretzdl*. Das Verzeichnis (Repository) in dem die Projekte liegen heißt */usr/local/cvs/bretzdl*. Zur Verbindung wird die *ssh* verwendet. Somit müssen die Umgebungsvariablen entsprechend nachfolgend gesetzt werden:

CVSROOT=:ext:bretzdl@rai16/usr/local/cvs/bretzdl

CVS_RSH=ssh

Es existieren folgende Projekte:

Leon-1: der Quelltext von LEON mit Buildskripten

cpld: die beiden CPLD Designs mit serieller Schnittstellen Unterstützung

xstools: der Quelltext der Linux Portierung der Xess-Software

ddm: das digitale Diktiergerät als Beispieldesign

xsvsoft: das Arbeitsverzeichnis der Xess-Software

manuals: bei der Arbeit verwendeten Dokumente und Handbücher

designs: synthetisierte Designs

downloads: Original Quelltexte und Programme

Anhang B

ModelSIM

Der Compiler von Mentor wird mit *vcom* für VHDL und *vlog* für Verilog Programmcode aufgerufen. Er benötigt für die Arbeit ein Verzeichnis, in dem er die Daten ablegen kann. Dieses wird mit dem Aufruf *vlib <Verzeichnisname>* erstellt. Zusätzlich existiert eine Datei *modelsim.ini,* in der die Pfade der Bibliotheken und sonstige Konfigurationsparameter für Mentor ModelSIM stehen. Um den Quelltext von LEON zu übersetzen, gibt es ein speziell für ModelSIM ausgelegtes Makefile. Durch den Aufruf von *make,* in dem Verzeichnis *leon,* entsteht in dem Unterverzeichnis *work* das Simulationsmodell.

Um LEON zu simulieren, muss ModelSIM durch den Aufruf *vsim* in dem Hauptverzeichnis gestartet werden. Nur durch Aufruf aus dem Hauptverzeichnis findet es das Verzeichnis *tsource* und die darin befindlichen Programmdateien. In dem Hauptverzeichnis befindet sich ebenfalls ein Makefile, welches die Makefiles in den Unterverzeichnissen ausführt. Damit können mit einem Aufruf alle Quellen übersetzt werden.

Durch Auswahl einer Testbench Konfiguration (tbdef, tb_32_0_32_0, usw.) wird das Design geladen und kann mit *run* gestartet werden.

Anhang C

Portierung der XESS Software nach LINUX

XESS liefert mit dem XSV-800 Board nur Software für Microsoft Windows. Für die Abteilung Rechnerarchitektur, die in ihren Rechnerpools nur Unix Maschinen betreibt, hätte dies die Installation eines Windowsrechners bedeutet. Glücklicherweise liefert XESS den Quelltext für ihre Software mit. Dadurch wurde es möglich, sie nach Linux zu portieren. Die Umsetzung erfolgte nur für die Eingabeaufforderung basierten Werkzeuge, also nicht für die mit grafischer Benutzungsoberfläche. Mit ihnen ist aber der volle Betriebsumfang des Boards möglich.

Der Quelltext lag in der Version 3.0 vor. Auf Grund der guten Portierbarkeit von C++ und C, so wie Vermeidung von Verwendung der MFC[1]in dem Quelltext, musste nur wenig angepasst werden. Der größte Teil bezieht sich dabei auf die Änderung der Verzögerungsschleifen für das Zeitverhalten, der Programmierung der parallelen Schnittstelle und Änderung des Programms *xsload*, so dass mit diesem auch exo-Datein in das FlashRAM geladen werden können.

C.1 Verzeichnisstruktur

Der Quelltext des Programms ist in dem Projekt *xstools* im CVS abgelegt. Der größte Teil befindet sich in dem Verzeichnis *xstools/XSTOOLS*. Die meisten Dateien sind nach der Klasse benannt, die sie beinhalten. In den Unterverzeichnissen befinden sich jeweils die Hauptprogrammdateien, die die Objekte aus dem oberliegenden Verzeichnis benutzen. Die Unterverzeichnisse tragen dabei den Namen der Anwendung, die in ihnen steht. Alle Verzeichnisse und Dateien, die mit '*gxs*' beginnen, haben eine grafische Oberfläche. Diese wurden nicht angepasst. Es

[1]Microsoft Foundation Classes

wurden lediglich die Dateien *xsload* und *xsport* in den Unterverzeichnissen verändert und übersetzt. Nach dem Übersetzen des Quelltextes stehen in diesen beiden Unterverzeichnissen auch die beiden ausführbaren Programme. Zur Übersetzung steht ein *Makefile* in dem Verzeichnis des Quelltextes zur Verfügung.

C.2 Änderungen des Quelltextes im Detail

Damit die parallele Schnittstelle unter Linux benutzt werden kann, musste das Modul zur Ansteuerung der Hardware angepasst werden. Die Xess Software greift zur Übertragung der Daten direkt auf die Konfigurationsregister der Hardware zu. Da unter Linux dieselbe Hardware verwendet wird, konnte der Teil der Software, der die Steuersignale erzeugt, beibehalten werden. Um Zugriff auf die I/O Schnittstellen der Hardware, unter Windows, zu bekommen, benutzt Xess Software eines Drittanbieters. Dieses Modul wurde durch eigene Software ersetzt und beschränkt sich auf ein paar Zeilen Code, welcher in der Datei *DLPORTIO.C* steht [IOM00]. Die I/O Adresse wurde fest auf den Bereich 0x3f8[2] gesetzt. Durch den Aufruf der Funktion *init()* wird der I/O Bereich vom Betriebssystem freigeschaltet. Dazu muss der ausführende Prozess Rootrechte haben. Dies kann durch Ausführen des Programms als root oder durch Setzen des Besitzers der Datei auf root plus gesetztem S-Bit geschehen. Nachdem der I/O Bereich freigeschaltet ist, werden die Rechte des Prozesses auf die userid des ausführenden Benutzers gesetzt.

Für das richtige Zeitverhalten musste die Methode *InsertDelay()* der Klasse *XCPort* in *XCPORT.CPP* an Linux angepasst werden. Unter Microsoft Windows liefert der Funktionsaufruf *clock()* die vergangene Zeit in Millisekunden seit dem Start des Programms. Unter Linux liefert *clock()* die vergangene Zeit seit dem letzten *clock()* Aufruf. Während unter Windows die Differenz benötigt wird, kann unter Linux direkt der zurückgegebene Wert verwendet werden.

In dem Kommandozeilenprogentsprechendenramm *xsload* wurde die Unterstützung von exo-Dateien eingebaut. Diese können von der Windows Version nur von den Programmen mit grafischer Oberfläche gehandhabt werden. Das Programm *xsload* in *xstools/xsload* wurde auf den Aufruf mit entsprechenden Datein als Parameter erweitert. Zur Handhabung der S-Records in den exo-Datein, musste die Klasse *XSV95* aus *XSV95.CPP* in das Programm aufgenommen werden und entsprechend verwendet werden.

Zur korrekten Übertragung von svf-Dateien musste in der Klasse *XC95* in *XC95.CPP* die Ausgabe der Strings, die Methode *data()* zu *c_str()* gewechselt werden. Da die C Funktion *sscanf()* verwendet wird, müssen die String Daten unter Linux in richtige C-Strings mit abschließendem '\0' Zeichen transformiert

[2] Standardadresse des Parallelport 0

werden.

In der Klasse *XSError* in *Xserror.CPP* wurde der Aufruf *ostream_withassign()* in den von Linux benötigten Aufruf *_IO_ostream_withassign()* geändert.

Die restlichen Änderungen betreffen hauptsächlich das Verschieben der Deklaration von Variablen im Programmtext oder ähnliche kleinere Anpassungen. Diese werden nicht näher erläutert.

Anhang D

Datei Auszüge

D.1 target.vhd

```
------------------------------------------------------------------------------
-- Synthesis configurations
------------------------------------------------------------------------------

constant syn_none  :  syn_config_type := (
  syntool => synplify, targettech => gen, infer_pads => true,
  infer_ram => false, infer_regf => false, infer_rom => false,
  gatedclk => false, rfsyncrd => true, rfsyncwr => true);
constant syn_atc35 :  syn_config_type := (
  syntool => synplify, targettech => atc35, infer_pads => false,
  infer_ram => false, infer_regf => false, infer_rom => true,
  gatedclk => false, rfsyncrd => true, rfsyncwr => true);
constant syn_synplify :  syn_config_type := (
  syntool => synplify, targettech => gen, infer_pads => true,
  infer_ram => true, infer_regf => true, infer_rom => true,
  gatedclk => false, rfsyncrd => true, rfsyncwr => true);
constant syn_synplify_vprom :  syn_config_type := (
  syntool => synplify, targettech => gen, infer_pads => true,
  infer_ram => true, infer_regf => true, infer_rom => false,
  gatedclk => false, rfsyncrd => true, rfsyncwr => true);
constant syn_leonardo :  syn_config_type := (
  syntool => leonardo, targettech => gen, infer_pads => true,
  infer_ram => true, infer_regf => true, infer_rom => true,
  gatedclk => false, rfsyncrd => true, rfsyncwr => true);
constant syn_virtex  :  syn_config_type := (
  syntool => synplify, targettech => virtex, infer_pads => true,
  infer_ram => false, infer_regf => false, infer_rom => true,
  gatedclk => false, rfsyncrd => true, rfsyncwr => true);
constant syn_virtex_vprom  :  syn_config_type := (
  syntool => synplify, targettech => virtex, infer_pads => true,
  infer_ram => false, infer_regf => false, infer_rom => false,
  gatedclk => false, rfsyncrd => true, rfsyncwr => true);

------------------------------------------------------------------------------
-- IU configurations
------------------------------------------------------------------------------

constant iu_std :  iu_config_type := (
  nwindows => 8, multiplier => iterative, fpuen => 0, cpen => false,
```

```
  fastjump => false, icchold => false, lddelay => 1, fastdecode => false,
  impl => 0, version => 0);
constant iu_fpga :  iu_config_type := (
  nwindows => 8, multiplier => none, fpuen => 0, cpen => false,
  fastjump => true, icchold => true, lddelay => 1, fastdecode => true,
  impl => 0, version => 0);

------------------------------------------------------------------------
-- FPU configurations
------------------------------------------------------------------------

constant fpu_none :  fpu_config_type := (fpu => none, fregs => 0, version => 0);
constant fpu_meiko:  fpu_config_type := (fpu => meiko, fregs => 32, version =>
0);
constant fpu_fpc  :  fpu_config_type := (fpu => fpc, fregs => 0, version => 0);

------------------------------------------------------------------------
-- CP configurations
------------------------------------------------------------------------

constant cp_none :  cp_config_type := (cp => none, version => 0);
constant cp_cpc  :  cp_config_type := (cp => cpc, version => 0);

------------------------------------------------------------------------
-- cache configurations
------------------------------------------------------------------------

constant cache_2k1k :  cache_config_type := (
  icachesize => 2, ilinesize => 4, dcachesize => 1, dlinesize => 4,
  bootcache => false);
constant cache_2k2k :  cache_config_type := (
  icachesize => 2, ilinesize => 4, dcachesize => 2, dlinesize => 4,
  bootcache => false);
constant cache_2kl8_2kl4 :  cache_config_type := (
  icachesize => 2, ilinesize => 8, dcachesize => 2, dlinesize => 4,
  bootcache => false);
constant cache_4k2k :  cache_config_type := (
  icachesize => 4, ilinesize => 8, dcachesize => 2, dlinesize => 4,
  bootcache => false);
constant cache_4k4k :  cache_config_type := (
  icachesize => 4, ilinesize => 4, dcachesize => 4, dlinesize => 4,
  bootcache => false);
constant cache_8k8k :  cache_config_type := (
  icachesize => 8, ilinesize => 8, dcachesize => 8, dlinesize => 4,
  bootcache => false);

------------------------------------------------------------------------
-- Memory controller configurations
------------------------------------------------------------------------

constant mctrl_std :  mctrl_config_type := (
  bus8en => true, bus16en => true, rawaddr => false);
constant mctrl_fpga :  mctrl_config_type := (
  bus8en => true, bus16en => true, rawaddr => false);
constant mctrl_mem32 :  mctrl_config_type := (
  bus8en => false, bus16en => false, rawaddr => false);
constant mctrl_bprom :  mctrl_config_type := (
  bus8en => false, bus16en => false, rawaddr => false);
constant mctrl_xess16 :  mctrl_config_type := (
  bus8en => false, bus16en => true, rawaddr => false);

------------------------------------------------------------------------
```

```
-- boot configurations
------------------------------------------------------------------------

constant boot_mem :  boot_config_type := (boot => memory, promabits => 1,
  ramrws => 0, ramwws => 0, sysclk => 20000000, baud => 38400, extbaud => false);
constant boot_prom :  boot_config_type := (boot => prom, promabits => 8,
  ramrws => 0, ramwws => 0, sysclk => 20000000, baud => 38400, extbaud=> false);
constant boot_cache :  boot_config_type := (boot => icache, promabits => 8,
  ramrws => 0, ramwws => 0, sysclk => 20000000, baud => 38400, extbaud=> false);
constant boot_prom_xess16 :  boot_config_type := (boot => prom, promabits => 8,
  ramrws => 0, ramwws => 0, sysclk => 20000000, baud => 38400, extbaud=> false);

------------------------------------------------------------------------
--  PCI configurations
------------------------------------------------------------------------

constant pci_none :  pci_config_type := (
  pcicore => none, cfgclk => mainclk, ahbmasters => 0, ahbslaves => 0);
constant pci_test :  pci_config_type := (
  pcicore => ahbtst, cfgclk => mainclk, ahbmasters => 2, ahbslaves => 1);
constant pci_insilicon :  pci_config_type := (
  pcicore => insilicon, cfgclk => both, ahbmasters => 2, ahbslaves => 1);
constant pci_estec :  pci_config_type := (
  pcicore => estec, cfgclk => mainclk, ahbmasters => 1, ahbslaves => 1);
constant pci_ahb_test :  pci_config_type := (
  pcicore => ahbtst, cfgclk => mainclk, ahbmasters => 0, ahbslaves => 1);

------------------------------------------------------------------------
--  Peripherals configurations
------------------------------------------------------------------------

constant peri_std :  peri_config_type := (
  cfgreg => true, ahbstat => true, wprot => true, wdog => true);
constant peri_fpga :  peri_config_type := (
  cfgreg => true, ahbstat => false, wprot => false, wdog => false);

------------------------------------------------------------------------
-- Debug configurations
------------------------------------------------------------------------

constant debug_none :  debug_config_type := ( enable => false, uart => false,
  iureg => false, fpureg => false, nohalt => false, pclow => 2);
constant debug_disas :  debug_config_type := ( enable => true, uart => false,
  iureg => false, fpureg => false, nohalt => false, pclow => 2);
constant debug_all  :  debug_config_type := ( enable => true, uart => true,
  iureg => true, fpureg => true, nohalt => false, pclow => 0);

------------------------------------------------------------------------
-- Amba AHB configurations
------------------------------------------------------------------------

-- standard slave config
constant ahbslvcfg_std :  ahb_slv_config_vector(0 to AHB_SLV_MAX-1) := (
-- first    last  index  split  enable  function           HADDR[31:28]
  ("0000", "0111",  0,   false, true), -- memory controller,   0x0- 0x7
  ("1000", "1000",  1,   false, true), -- APB bridge, 128 MB   0x8- 0x8
   others => ahb_slv_config_void);

-- AHB test slave config
constant ahbslvcfg_test :  ahb_slv_config_vector(0 to AHB_SLV_MAX-1) := (
-- first    last  index  split  enable  function           HADDR[31:28]
  ("0000", "0111",  0,   false, true), -- memory controller,   0x0- 0x7
```

```
  ("1000", "1000",  1,   false, true), -- APB bridge, 128 MB   0x8- 0x8
  ("1010", "1010",  2,   true,  true), -- AHB test module      0xA- 0xA
  ("1100", "1111",  3,   false, true), -- PCI initiator        0xC- 0xF
   others => ahb_slv_config_void);

-- PCI slave config
constant ahbslvcfg_pci :  ahb_slv_config_vector(0 to AHB_SLV_MAX-1) := (
-- first    last  index  split  enable  function           HADDR[31:28]
  ("0000", "0111",  0,   false, true), -- memory controller,   0x0- 0x7
  ("1000", "1000",  1,   false, true), -- APB bridge, 128 MB   0x8- 0x8
  ("1010", "1111",  2,   false, true), -- PCI initiator        0xA- 0xF
   others => ahb_slv_config_void);

-- standard cacheability config
constant ahbcachecfg_std :  ahb_cache_config_vector(0 to AHB_CACHE_MAX-1) := (
-- first    last        function   HADDR[31:29]
  ("000", "000"),   -- PROM area    0x0- 0x0
  ("010", "011"),   -- RAM area     0x2- 0x3
   others => ahb_cache_config_void);

-- standard config record
constant ahb_std :  ahb_config_type := (
  masters => 1, defmst => 0, split => false,
  slvtable => ahbslvcfg_std, cachetable => ahbcachecfg_std);
-- FPGA config record
constant ahb_fpga :  ahb_config_type := (
  masters => 1, defmst => 0, split => false,
  slvtable => ahbslvcfg_std, cachetable => ahbcachecfg_std);
-- Phoenix PCI core config record (uses two AHB master instefaces)
constant ahb_insilicon_pci :  ahb_config_type := (
  masters => 3, defmst => 0, split => false,
  slvtable => ahbslvcfg_pci, cachetable => ahbcachecfg_std);
-- ESTEC PCI core config record (uses one AHB master insteface)
constant ahb_estec_pci :  ahb_config_type := (
  masters => 2, defmst => 0, split => false,
  slvtable => ahbslvcfg_pci, cachetable => ahbcachecfg_std);
-- AHB test config
constant ahb_test :  ahb_config_type := (
  masters => 3, defmst => 0, split => true,
  slvtable => ahbslvcfg_test, cachetable => ahbcachecfg_std);

------------------------------------------------------------------------------
-- Amba APB configurations
------------------------------------------------------------------------------

-- standard config
constant apbslvcfg_std :  apb_slv_config_vector(0 to APB_SLV_MAX-1) := (
--   first         last      index  enable     function           PADDR[9:0]
( "0000000000", "0000001000",  0,   true), -- memory controller, 0x00 - 0x08
( "0000001100", "0000010000",  1,   true), -- AHB status reg.,   0x0C - 0x10
( "0000010100", "0000011000",  2,   true), -- cache controller,  0x14 - 0x18
( "0000011100", "0000100000",  3,   true), -- write protection,  0x1C - 0x20
( "0000100100", "0000100100",  4,   true), -- config register,   0x24 - 0x24
( "0001000000", "0001101100",  5,   true), -- timers,            0x40 - 0x6C
( "0001110000", "0001111100",  6,   true), -- uart1,             0x70 - 0x7C
( "0010000000", "0010001100",  7,   true), -- uart2,             0x80 - 0x8C
( "0010010000", "0010011100",  8,   true), -- interrupt ctrl     0x90 - 0x9C
( "0010100000", "0010101100",  9,   true), -- I/O port           0xA0 - 0xAC
  others => apb_slv_config_void);

-- PCI config
constant apbslvcfg_pci :  apb_slv_config_vector(0 to APB_SLV_MAX-1) := (
```

```
--   first         last      index  enable     function           PADDR[9:0]
( "0000000000", "0000001000",  0,   true), -- memory controller, 0x00 - 0x08
( "0000001100", "0000010000",  1,   true), -- AHB status reg.,   0x0C - 0x10
( "0000010100", "0000011000",  2,   true), -- cache controller,  0x14 - 0x18
( "0000011100", "0000100000",  3,   true), -- write protection,  0x1C - 0x20
( "0000100100", "0000100100",  4,   true), -- config register,   0x24 - 0x24
( "0001000000", "0001101100",  5,   true), -- timers,            0x40 - 0x6C
( "0001110000", "0001111100",  6,   true), -- uart1,             0x70 - 0x7C
( "0010000000", "0010001100",  7,   true), -- uart2,             0x80 - 0x8C
( "0010010000", "0010011100",  8,   true), -- interrupt ctrl     0x90 - 0x9C
( "0010100000", "0010101100",  9,   true), -- I/O port           0xA0 - 0xAC
( "0100000000", "0111111100", 10,   true), -- PCI configuration  0x100- 0x1FC
  others => apb_slv_config_void);

constant apb_std :  apb_config_type := (table => apbslvcfg_std);
constant apb_pci :  apb_config_type := (table => apbslvcfg_pci);

------------------------------------------------------------------------------
-- Pre-defined LEON configurations
------------------------------------------------------------------------------

-- standard simulation
constant sim_std :  config_type := (
  synthesis => syn_none, iu => iu_std, fpu => fpu_none, cp => cp_none,
  cache => cache_2k2k, ahb => ahb_std, apb => apb_std, mctrl => mctrl_std,
  boot => boot_mem, debug => debug_disas, pci => pci_none, peri => peri_std);

-- simulatiom with Insilicon PCI core
constant sim_insilicon_pci :  config_type := (
  synthesis => syn_none, iu => iu_std, fpu => fpu_none, cp => cp_none,
  cache => cache_2k2k, ahb => ahb_insilicon_pci, apb => apb_pci,
  mctrl => mctrl_std, boot => boot_mem, debug => debug_disas,
  pci => pci_insilicon, peri => peri_std);

-- use AHB test module
constant sim_ahb_test :  config_type := (
  synthesis => syn_none, iu => iu_std, fpu => fpu_none, cp => cp_none,
  cache => cache_2k2k, ahb => ahb_test, apb => apb_pci, mctrl => mctrl_std,
  boot => boot_mem, debug => debug_disas, pci => pci_ahb_test,
  peri => peri_std);

-- synthesis using synplify, 2 + 2 Kbyte cache
constant synplify_2k2 :  config_type := (
  synthesis => syn_synplify, iu => iu_fpga, fpu => fpu_none, cp => cp_none,
  cache => cache_2k2k, ahb => ahb_fpga, apb => apb_std, mctrl => mctrl_fpga,
  boot => boot_mem, debug => debug_disas, pci => pci_none, peri => peri_fpga);

-- synthesis using synplify, internal boot prom (soft)
constant synplify_2k2k_softprom :  config_type := (
  synthesis => syn_synplify, iu => iu_fpga, fpu => fpu_none, cp => cp_none,
  cache => cache_2k2k, ahb => ahb_fpga, apb => apb_std, mctrl => mctrl_fpga,
  boot => boot_prom, debug => debug_disas, pci => pci_none, peri => peri_fpga);

-- synthesis using synplify, internal boot prom (virtex only)
constant synplify_2k2k_virtexprom :  config_type := (
  synthesis => syn_synplify_vprom, iu => iu_fpga, fpu => fpu_none, cp => cp_none,
  cache => cache_2k2k, ahb => ahb_fpga, apb => apb_std, mctrl => mctrl_fpga,
  boot => boot_prom, debug => debug_disas, pci => pci_none, peri => peri_fpga);

-- synthesis using leonardo, 2 + 2 Kbyte cache
constant leonardo_2k2 :  config_type := (
  synthesis => syn_leonardo, iu => iu_fpga, fpu => fpu_none, cp => cp_none,
```

```
  cache => cache_2k2k, ahb => ahb_fpga, apb => apb_std, mctrl => mctrl_fpga,
  boot => boot_mem, debug => debug_disas, pci => pci_none, peri => peri_fpga);

-- synthesis using leonardo, internal boot prom (soft)
constant leonardo_2k2k_softprom :  config_type := (
  synthesis => syn_leonardo, iu => iu_fpga, fpu => fpu_none, cp => cp_none,
  cache => cache_2k2k, ahb => ahb_fpga, apb => apb_std, mctrl => mctrl_fpga,
  boot => boot_prom, debug => debug_disas, pci => pci_none, peri => peri_fpga);

-- synthesis for VIRTEX, any syntool
constant gen_virtex_2k2k :  config_type := (
  synthesis => syn_virtex, iu => iu_fpga, fpu => fpu_none, cp => cp_none,
  cache => cache_2k2k, ahb => ahb_fpga, apb => apb_std, mctrl => mctrl_std,
  boot => boot_mem, debug => debug_disas, pci => pci_none, peri => peri_fpga);

-- synthesis for VIRTEX, any syntool, xess16
constant gen_virtex_2k2k_xess16 :  config_type := (
  synthesis => syn_virtex, iu => iu_fpga, fpu => fpu_none, cp => cp_none,
  cache => cache_2k2k, ahb => ahb_fpga, apb => apb_std, mctrl => mctrl_xess16,
  boot => boot_mem, debug => debug_disas, pci => pci_none, peri => peri_fpga);

-- synthesis for VIRTEX, any syntool, soft boot prom
constant gen_virtex_2k2k_bprom :  config_type := (
  synthesis => syn_virtex, iu => iu_fpga, fpu => fpu_none, cp => cp_none,
  cache => cache_2k2k, ahb => ahb_fpga, apb => apb_std, mctrl => mctrl_bprom,
  boot => boot_prom, debug => debug_disas, pci => pci_none, peri => peri_fpga);

-- synthesis for VIRTEX, any syntool, soft boot prom, xess16
constant gen_virtex_2k2k_bprom_xess16 :  config_type := (
  synthesis => syn_virtex, iu => iu_fpga, fpu => fpu_none, cp => cp_none,
  cache => cache_2k2k, ahb => ahb_fpga, apb => apb_std, mctrl => mctrl_xess16,
  boot => boot_prom_xess16, debug => debug_disas, pci => pci_none, peri =>
peri_fpga);

-- synthesis for VIRTEX, any syntool, hard boot prom
constant gen_virtex_2k2k_vprom :  config_type := (
  synthesis => syn_virtex_vprom, iu => iu_fpga, fpu => fpu_none, cp => cp_none,
  cache => cache_2k2k, ahb => ahb_fpga, apb => apb_std, mctrl => mctrl_bprom,
  boot => boot_prom, debug => debug_disas, pci => pci_none, peri => peri_fpga);

-- synthesis targetting ATC35 asic lib, any syntools
constant gen_atc35 :  config_type := (
  synthesis => syn_atc35, iu => iu_std, fpu => fpu_fpc, cp => cp_none,
  cache => cache_4k4k, ahb => ahb_std, apb => apb_std, mctrl => mctrl_std,
  boot => boot_mem, debug => debug_disas, pci => pci_none, peri => peri_std);
end;
```

D.2 LEON Top-Entity

```
entity leon is
  port (
    resetn :  in std_logic; -- system signals
    clk :  in std_logic;
    errorn :  out std_logic;
    address :  out std_logic_vector(27 downto 0); -- memory bus
    data :  inout std_logic_vector(31 downto 0);
    ramsn :  out std_logic_vector(3 downto 0);
    ramoen :  out std_logic_vector(3 downto 0);
    rwen :  inout std_logic_vector(3 downto 0);
```

```
    romsn :  out std_logic_vector(1 downto 0);
    iosn :  out std_logic; oen :  out std_logic;
    read :  out std_logic; writen :  inout std_logic;
    brdyn :  in std_logic; bexcn :  in std_logic;
    pio :  inout std_logic_vector(15 downto 0); -- I/O port
    wdogn :  out std_logic; -- watchdog output
    test :  in std_logic
  );
end;
```

D.3 ddm.h

```
#ifndef __ASSEMBLER__
struct ddmregs {
  volatile unsigned int controlreg;  /* 0x00 */
  volatile unsigned int startaddr;
  volatile unsigned int stopaddrr;
  volatile unsigned int scalerupr;
  volatile unsigned int displcontr;  /* 0x10 */
  volatile unsigned int buttonreg;
  volatile unsigned int act_mem_adr
};
#define REGSTART 0x80000200
#endif
```

Literaturverzeichnis

[ARM99] ARM Limited: *AMBA Specification*. 2.0. 1999

[ASA97] ASAHI KASEI Microsystems (AKM): *AK4520A Description*. March 1997

[Ash96] Kap. 18 In: ASHENDEN, Peter J.: *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers, 1996, S. 511. – ISBN 1–55860–270–4

[Bou00a] BOUT, D. V.: *XSV Flash Programming and Virtex Configuration*. 1.0. XESS Corporation, April 2000

[Bou00b] BOUT, D. V.: *XSV Parallel Port Interface*. 1.0. XESS Corporation, April 2000

[Fog99] FOGEL, Karl: *Open Source Development with CVS*. CoriolisOpen Press, 1999

[Gai99] GAISLER, Jiri: *GNU Cross-Compiler System*. 2.0.7. ESA/ESTEC, November 1999

[Gai00a] GAISLER, Jiri: *LEON-1 VHDL model description*. 2.2. ESA/ESTEC, October 2000

[Gai00b] GAISLER, Jiri: *MKProm*. 1.2.7. ESA/ESTEC, 2000

[Gai00c] GAISLER, Jiri: *SIS manual*. 3.0.5. ESA/ESTEC, 2000

[IOM00] *I/O Port Programming mini-HOWTO*. `http://www.how2linux.com`. November 2000

[Jen94] JENKINS, Jesse H.: *Designing with FPGAs and CPLDs*. Prentice Hall, 1994. – ISBN 0–13–721549–5

[leo01] *LEON Homepage*. `http://www.estec.esa.nl/wsmwww/leon`. 2001

[met01] *Metaflow Homepage*. `http://www.metaflow.com`. 2001

[MK98] MICHAEL KEATING, Pierre B.: *Reuse Methology Manual*. Kluwer Academic Publishers, 1998. – ISBN 0–7923–8175–0

[On-98] On-Line Applications Research Corporation: *RTEMS C User's Guide*. 4.0.0. October 1998

[RHP93] ROLAND H. PESCH, Jeffrey M. O.: *The GNU Binary Utilities*. 2.9.1. Cygnus, May 1993

[RMS98] RICHARD M. STALLMAN, Roland H. P.: *Debugging with GDB*. 5.0. Free Software Foundation, Inc., April 1998

[SPA92] SPARC International, Inc: *The SPARC Architecture Manual Version 8*. 1992

[Sta98] STALLMAN, Richard M.: *Using and Porting GNU CC*. for egcs-1.1. Free Software Foundation, Inc., March 1998

[VSI97] VSI Alliance: *VSI Alliance Architecture Document*. 1.0. 1997

[XES00] XESS Corporation: *XSV Board Manual*. 1.0. March 2000

[xes01] *Xess Homepage*. `http://www.xess.com`. 2001

[Xil98] Xilinx: *XC95108 In-System-Programmable CPLD*. 3.0. December 1998

[Xil00a] Xilinx: *Alliance Series 3.1i Quick Start Guide*. 2000. – Seite 1-5

[Xil00b] Xilinx: *Using the Virtex Block SelectRAM+ Features*. 1.3. March 2000

[Xil00c] Xilinx: *Virtex 2.5 V Field Programmable Gate Arrays*. 2.3. September 2000

[xil01] *Xilinx Homepage*. `http://www.xilinx.com`. 2001

Index

AHB, 29
AHB Master, 54
AK4520A, 49
AMBA, 27
APB, 29
APB Slave, 50
Arbiter, 30
ARC, 11
ARM, 11
ASB, 28

Befehlscache, 14
bit-Datei, 42
bitgen, 22

CLB, 35
clear_buff(), 58
CPLD, 35, 36
CVS, 63
CVSROOT, 63

Datencache, 14
ddm(), 58
DDM-Register, 56
ddm.c, 55
ddm.h, 55
ddm_play(), 59
ddm_record(), 58
Designflow, 21
dispoff(), 57
dsgnmgr, 22

eingebettetes System, 7
ERC32, 11
exo-Datei, 43

Firmcore, 8
FPGA, 9, 35
free_block(), 57
Freedom, 11

GAL, 35
gdb, 26
geistiges Eigentum, 8
get_block(), 56
get_buttons(), 57

Hardcore, 8
hexddisp(), 57

intellectual property, 8
IP, 8

LEON, 11

map, 22
memory mapped register, 13
minicom, 43
mkprom, 27
ModelSIM, 18

ngd2vhdl, 22
ngdanno, 22
ngdbuild, 22

OpenRISC, 11

PAL, 35
par, 22
Personalisierung, 35
PLA, 35
PLD, 9, 35

Programmierung, 26
promgen, 22

Registerwindows, 14

S-Record, 27, 44
Scalable Processor Architecture, 12
serielle Schnittstelle, 43
set_sample_freq(), 57
seyon, 43
Simulation Gatterebene, 22
Simulation Registertransferebene, 18
SIS, 26
sis, 26
SoC, 7
Softcore, 8
SPARC, 12
sparc-rtems-gcc, 26
sparc-rtems-objcopy, 27, 44
sparc-rtems-objdump, 27
Speicher abgebildete Register, 13
startplay(), 57
startrec(), 57
stop(), 57
structure ddmregs, 56
syn, 15, 21
Synthese, 20
System-on-a-Chip, 7

target.vhd, 25
tbench, 15, 18
Tensilica, 11
tenv, 16, 20
tenv32_back, 16, 20
test(), 58
Testbench, 19
trans(), 57
trans_back(), 57
tsource, 16, 18

vcom, 67
Verzeichnisse, 15
Virtual Socket Interface Alliance, 8
vlib, 67
vlog, 67
volatile, 56
VSIA, 8

wait_for_audiofinish(), 57

XC95108, 36
XCheckeranschluss, 37
XCV800, 35
XESS, 33
xsload, 42
xsport, 42
XSTOOLS_BIN_DIR, 42
XSV-800, 33
XSV-800 Peripherie, 34

Ich versichere, dass ich diese Arbeit selbständig verfasst und nur die angegebenen Hilfsmittel verwendet habe.

Daniel Bretz